Dr. Dobb's Journal of

Software Tools

FOR THE PROFESSIONAL PROGRAMMER

111 JANUARY 1986
$2.95 (3.95 CANADA)

Launching Our Second Decade

Bringing up a 68K Machine from Scratch

PL/68K: Is it C or is it Assembler?

An 8080 Simulator for the 68K

DOS Shell Notes on 80286 Big DOS

0 38351 16562

H.E.L.P. Eliminates Every Bug known to Compilers . . . As well as a few other species

H.E.L.P. is a completely interactive C programming environment with three innovative full-sized features that will revolutionize the way you write code.

A Clean Compile — Guaranteed!

Say Good-bye to all compiler-type errors. **H.E.L.P.** 's built-in program checker (which would embarrass LINT) not only hunts down bugs . . . explains the proper syntax . . . gives examples of usage . . . but will even offer suggested corrections. If you want, **H.E.L.P.** will even make the corrections for you . . . at the touch of a key.

H.E.L.P. also finds semantic errors as well as poor style and inefficiencies. You can even check the portability of your code!

Multi-Window Editing

Open as many windows as you want . . . . there's no limitation . . . not even your own memory. Because **H.E.L.P.** uses a virtual memory system you can create programs larger than your machine capacity.

H.E.L.P.'s very powerful editor allows you the flexibility to work in several windows . . . with several files at the same time.

Circle no. **165** on reader service card.

Save Keystrokes

Hundreds of commands are bound to the keyboard to give you fast execution.

Always be in Control

Not only can you develop code in many windows at the same time, but you can show (and refer to) important definitions in one window while creating in another. Or open a window and keep notes about your program . . . or type a memo . . . or a letter.

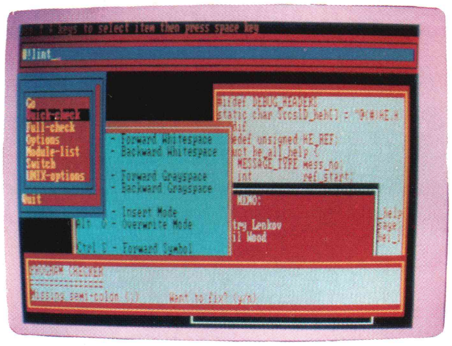
Increase your Productivity by 300% or More . . . . .

If you are a novice programmer, you'll begin writing code like an advanced programmer much faster with **H.E.L.P.**

Just imagine what **H.E.L.P.** will do for the ADVANCED PROGRAMMER.

You'll have more time to become creative with your algorithm (since **H.E.L.P.** will make sure your code compiles the "first time at bat").

H.E.L.P. tracks every step you make. If you are not sure about a command, just press a key, and you'll get the kind of help you need.



Check These Features

- *Multi-window environment*
- *Interactive program checking*
- *Check syntax, semantic, type usage, intermodule inconsistencies and portability*
- *Multi-file editing*
- *Intelligent help sub-system*
- *User-definable key-board bindings*
- *Supports color and monochrome*
- ***H.E.L.P.** supports the full C Language*

NOW IN MS-DOS

Order now $395

EVEREST SOLUTIONS

Everest Solutions, Inc.
3350 Scott Boulevard
Building 58
Santa Clara, CA 95051
(408) 986-8977

CONTENTS

Dr. Dobb's Journal of

Software Tools

FOR THE PROFESSIONAL PROGRAMMER

ARTICLES

Is it C or is it assembly? WordStar on an Atari ST? ▶

COMPILERS: PL/68K *by Edward K. Ream* **26**
The author of the RED editor could find no acceptable compiler for the 68K, so he wrote one. But PL/68K is not like other compilers.

SYSTEMS: A Simple OS for Real Time Applications *by Nick Turner* **44**
In setting up a multiuser system, the author found ways to shave many machine cycles off every instruction. Here he explains how he did it and offers some illustrative code.

HARDWARE: Bringing up the 68000 *by Alan K. Wilcox* **60**
Here's how to get a 68000 running very quickly.

PORTABILITY: COM: An 8080 Simulator for the MC68000 *by Jim Cathey* **76**
Besides being a useful tool, this program contains some insights gained in the process of developing a 68K application. The author also shows how to extend the simulator for Z80 instructions.

COLUMNS

Mysterious and dangerous programs ▶

C CHEST: A Unix-like Shell for MS DOS *by Allen Holub* **18**
Allen describes how the shell works on a high level and includes examples of some often-used functions.

16-BIT TOOLBOX: Trojan Horse Programs *by Ray Duncan* **118**
These mysterious and destructive programs are appearing on bulletin boards. What can be done to combat them?

FORUM

What C programmers don't know can hurt them. ▶

DDJ is now on CompuServe. ▶

EDITORIAL: The New Look *by Michael Swaine* **6**

LETTERS: Comment *by our readers* **8**

CARTOON: A Word on Formats *by Rand Renfroe* **8**

VIEWPOINT: Inefficient C *by Hal Hardenberg* **16**

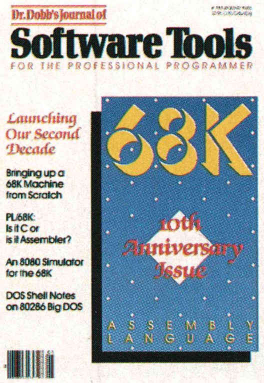
DDJ ON LINE: The Electronic Edition *by Frank DeRose* **17**

PROGRAMMERS' SERVICES

PROFESSIONAL PROGRAMMER: A little library of books on the profession **124**

OF INTEREST: New products of interest to programmers **126**

ADVERTISER INDEX: Where to find that ad **128**



Our anniversary cover was designed by Shelley Rae Doeden, who also is responsible for the new look of the magazine. Shelley is *DDJ's* Art Director.

This Issue:
The Motorola 68000 chip is, some say, a programmer's processor. The instruction set is rich and logically constructed, and memory access is simple and capacious. What it lacks, others say, are a standard operating system and development tools. This month we focus on programming tools for the 68K.

Next Issue:
Next month we'll look at structured programming and at some languages that have a reputation for encouraging structured design. We'll publish programming tools in Pascal, Modula-2, and Ada. We'll tell you where to get public-domain Ada utilities to speed Ada software development, and we'll look at a program that ports between dialects of Pascal.

PERFORMANCE PACKAGE

Blaise Computing Inc. introduces the PERFORMANCE PACKAGE™ for Turbo Pascal programmers.

TURBO PASCAL

Turbo ASYNCH™

With Turbo ASYNCH, you can be in constant touch with the world without ever leaving the console. Rapid transit at its best. Turbo ASYNCH is designed to let you incorporate asynchronous communication capabilities into your Turbo Pascal application programs, and it will drive any asynchronous device via the RS232 ports, like printers, plotters, modems or even other computers. Turbo ASYNCH is fast, accurate and lives up to its specs. Features include…

◆ Initialization of the COM ports allowing you to set all transmission options. ◆ Interrupt processing. ◆ Data transfer between circular queues and communications ports. ◆ Simultaneous buffered input and output to both COM ports. ◆ Transmission speeds up to 9600 Baud. ◆ Input and output queues as large as you wish. ◆ XON/XOFF protocol.

The underlying functions of Turbo ASYNCH are carefully crafted in assembler for efficiency, and drive the UART and programmable interrupt controller chips directly. These functions, installed as a runtime resident system, require just 3.2K bytes. The interface to the assembler routines is written in Turbo Pascal.

The Turbo Pascal PERFORMANCE PACKAGE™ is for the serious Turbo Pascal programmer who wants quality tools to develop applications. Every system comes with a comprehensive User Reference Manual, all source code and useful sample programs. They require an IBM PC or compatible, utilizing MS-DOS version 2.0 or later. There are no royalties for incorporating PERFORMANCE PACKAGE functions into your applications.

Turbo POWER TOOLS and Turbo ASYNCH sell for $99.95 each, and they may be ordered directly from Blaise Computing Inc. To order, call (415) 540-5441.

NEW!

Turbo POWER TOOLS™

Turbo POWER TOOLS is a sleek new series of procedures designed specifically to complement Turbo Pascal on IBM and compatible computers. Every component in Turbo POWER TOOLS is precision engineered to give you fluid and responsive handling, with all the options you need packed into its clean lines. High performance and full instrumentation, including…

◆ Extensive string handling to complement the powerful Turbo Pascal functions. ◆ Screen support and window management, giving you fast direct access to the screen without using BIOS calls. ◆ Access to BIOS and DOS services, including DOS 3.0 and the IBM AT. ◆ Full program control by allowing you to execute any other program from within your Turbo Pascal application. ◆ Interrupt service routines written entirely in Turbo Pascal. Assembly code is not required even to service hardware interrupts like the keyboard or clock.

Using Turbo POWER TOOLS, you can now "filter" the keyboard or even DOS, and create your own "sidekickable" applications.

YES, send me the Best for the Best! Enclosed is ________ for
☐ Turbo ASYNCH ☐ Turbo POWER TOOLS. (CA residents add 6½% Sales Tax. All orders add $6.00 for shipping.)

Name: ________ **Phone:** ________
Shipping Address: ________
City: ________ **State:** ________ **Zip:** ________
VISA or MC #: ________ **Exp. Date:** ________

Turbo Pascal is a trademark of Borland International. Turbo POWER TOOLS, Turbo ASYNCH and PERFORMANCE PACKAGE are trademarks of Blaise Computing Inc. IBM is a registered trademark of Internationl Business Machines Corporation. MS-DOS is a trademark of Microsoft Corporation.

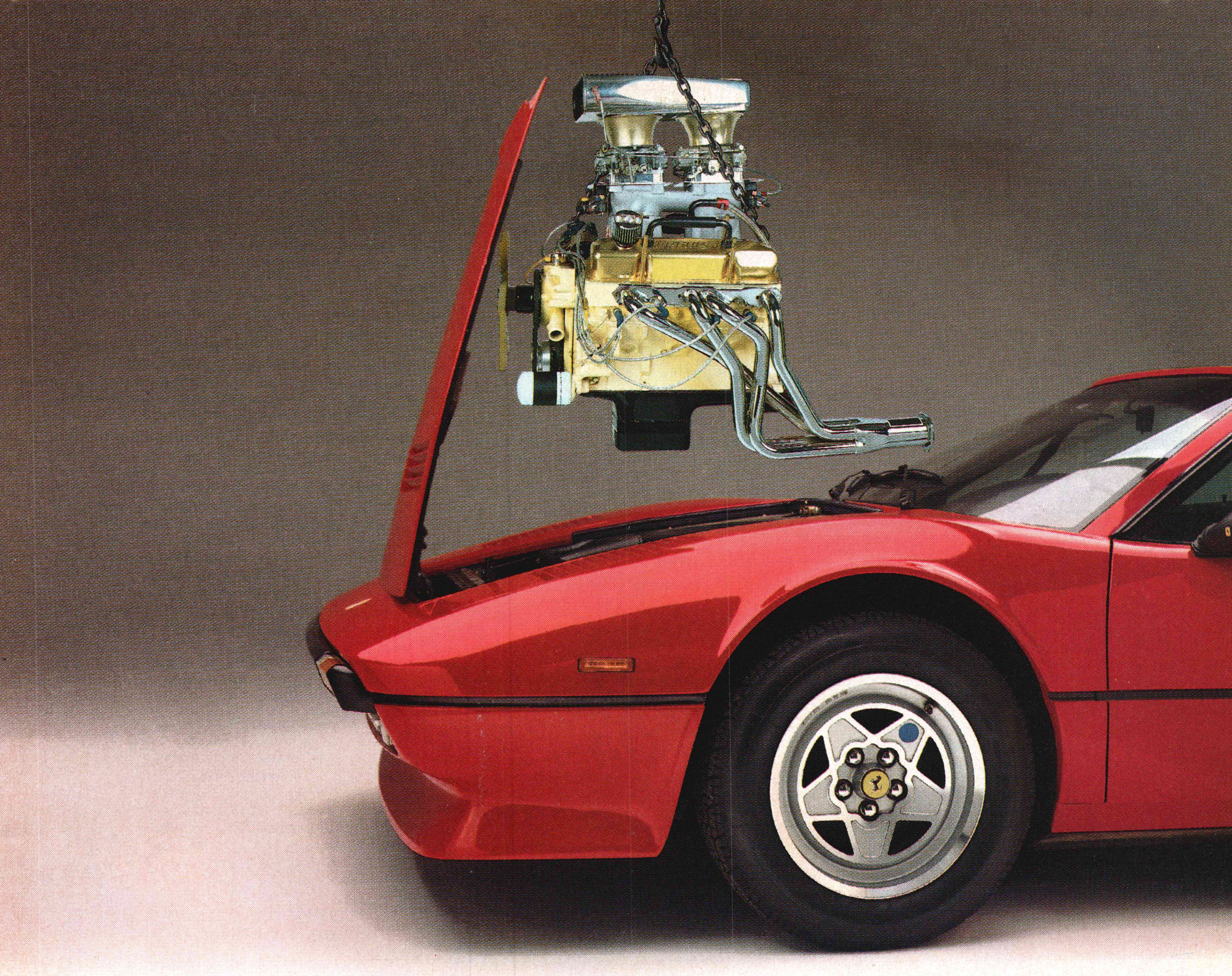
BLAISE COMPUTING INC.

BLAISE

watch us!

◆ 2034 BLAKE STREET
◆ BERKELEY, CA 94704
◆ (415) 540-5441

Circle **no. 159** on reader service card.

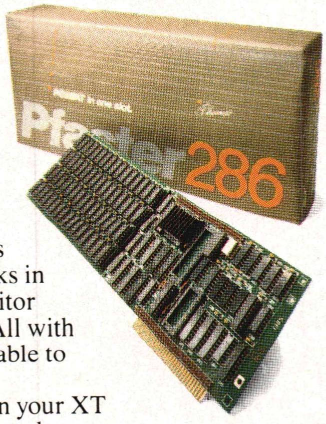


AT™ Pfantasies for your PC or XT.™

Want better speed and memory on your PC or XT without buying an AT?

You've got it!

Phoenix's new Pfaster™286 co-processor board turns your PC or XT into a high-speed engine 60 percent faster than an AT. Three times faster than an XT. It even supports PCs with third-party hard disks. But that's only the beginning.



You can handle spreadsheets and programs you never thought possible. Set up RAM disks in both 8088 and 80286 memory for linkage editor overlays or super-high-speed disk caching. All with Pfaster286's 1mb of standard RAM, expandable to 2mb, and dual-mode design.

You can develop 8086/186/286 software on your XT faster. Execute 95 percent of the application packages that run on the AT, excluding those that require fancy I/O capabilities your PC or XT hardware just isn't designed to handle. Queue multi-copy, multi-format print jobs for spooling. Or, switch to native 8088 mode to handle hardware-dependent programs and back again without rebooting. All with Pfaster286's compatible ROM software. And, Pfaster286 does the job unintrusively! No motherboard to exchange. No wires to solder. No chips to pull. Just plug it into a standard card slot, and type the magic word, "PFAST."

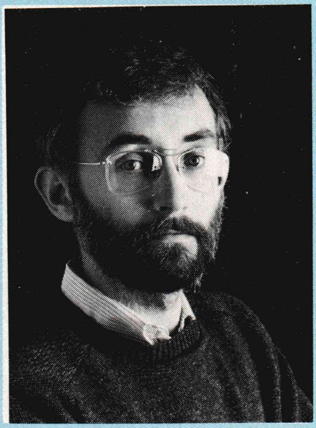
If you really didn't want an AT in the first place, just what it could do for you, call or write: Phoenix Computer Products Corp., 320 Norwood Park South, Norwood, MA 02062; (800) 344-7200. In Massachusetts, 617-762-5030.

Programmers' Pfantasies™
by
Phoenix

XT and AT are trademarks of International Business Machines Corporation. Pfaster286 and Programmers' Pfantasies are trademarks of Phoenix Computer Products Corporation.
For the Ferrari aficionado: yes, we know this is a rear engine car. We are showing the addition of a second engine to symbolize how Pfaster can be added to your PC or XT to increase performance.

Circle **no. 91** on reader service card.

EDITORIAL



Late 1975 and early 1976 saw a number of new magazines for a new group of readers: the pioneering users of the first microcomputers. Most were heavy on hardware because there was essentially no software to write about back then. In January 1976, a remedy for this situation appeared in the form of a magazine providing the tools needed to develop software: *Dr. Dobb's Journal of Tiny BASIC Calisthenics & Orthodontia*.

Ten years later, *DDJ* is still providing software tools for serious programmers. We've replaced "calisthenics and orthodontia" with other metaphors and have moved upscale from a black-and-white newsletter to a four-color magazine, but we still publish the only pages in which you can find compilers, assemblers, and programming tools reviewed, designed, analyzed, and source-listed, with the whole process watched over by the most knowledgable readership in the industry.

Was that last paragraph excessively self-congratulatory? Pardon it, but there is occasion for it: This month is *DDJ*'s tenth birthday. This birthday issue features articles on 68000 programming, leading off with Edward Ream's assembler that thinks it's a compiler. In a more overtly festive act, this issue also unveils the new design for *DDJ*.

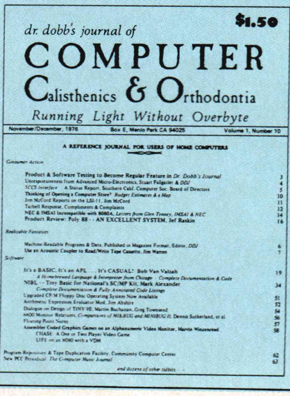
The new design should more directly indicate what the magazine provides: programming tools in the form of articles and listings, reviews, columns, a forum for discussion of programming issues, and services such as product listings. The title treatment, while harking back to the original cover design, makes it clear that what this magazine contains are software tools. Inside, we've gathered all the contents into five sections: Articles, Reviews, Columns, Forum, and Programmers' Services. Forum contains the editorial, letters, an editorial cartoon, and an invited viewpoint—this month by 68K authority Hal Hardenberg. Programmers' Services will grow; this month it contains a section of information on the professional side of programming. No room for reviews this month.

We are fortunate to have some excellent columnists who provide a personal view along with useful tips and utilities. Unfortunately, two of them are retiring from their columnist duties: Dave Cortesi and Bob Blum. Bob is just too busy to keep up a monthly column and to do as good a job as he wants; he'll continue to contribute articles and to maintain his CP/M bulletin board. He has written a sign-off letter that appears on page 8. Dave is leaving computer writing for the writing of fiction. Both were longtime contributors and will be missed. We are actively, if grudgingly, seeking replacements. Perhaps you know a candidate?

Finally, there's the electronic component we've added to the magazine. By the time you read this, *DDJ* should be up on CompuServe. See page 17 for details.

Michael Swaine

Michael Swaine



$1.50

dr. dobb's journal of

COMPUTER Calisthenics & Orthodontia

Running Light Without Overbyte

November/December, 1976 — Box E, Menlo Park CA 94025 — Volume 1, Number 10

A REFERENCE JOURNAL FOR USERS OF HOME COMPUTERS

Consumer Action

Product & Software Testing to Become Regular Feature in *Dr. Dobb's Journal*	3
Unresponsiveness from Advanced Micro-Electronics, Stuart Fallgatter & *DDJ*	4
SCCS Interface A Status Report, Southern Calif. Computer Soc. Board of Directors	5
Thinking of Opening a Computer Store? *Budget Estimates & a Map*	10
Jim McCord Reports on the LSI-11, Jim McCord	11
Tarbell Response, Compliments & Complaints	12
NEC & IMSAI Incompatible with 8080A, *Letters from Glen Tenney, IMSAI & NEC*	14
Product Review: Poly 88 - - AN EXCELLENT SYSTEM, Jef Raskin	16

Realizable Fantasies

Machine-Readable Programs & Data, Published in Magazine Format, Editor, *DDJ*	6
Use an Acoustic Coupler to Read/Write Tape Cassette, Jim Warren	7

Software

It's a BASIC, It's an APL . . . It's CASUAL! Bob Van Valzah *A Homebrewed Language & Interpreter from Chicago – Complete Documentation & Code*	19
NIBL - - Tiny Basic for National's SC/MP Kit, Mark Alexander *Complete Documentation & Fully Annotated Code Listings*	34
Upgraded CP/M Floppy Disc Operating System Now Available	51
Arithmetic Expression Evaluator Mod, Jim Abshire	52
Dialogue on Design of TINY HI, Martin Buchanan, Greg Townsend	54
6800 Monitor Relations, *Comparisons of MIKBUG and MINIBUG II,* Dennis Sutherland, et al.	56
Floating Point Notes	57
Assembler Coded Graphics Games on an Alphanumeric Video Monitor, Marvin Winzenread CHASE A One or Two Player Video Game; LIFE on an 8080 with a VDM	58
Program Repository & Tape Duplication Facility, Community Computer Center	62
New PCC Periodical: *The Computer Music Journal*	63

and dozens of other tidbits - -

Dr. Dobb's Journal of

Software Tools

Editorial

Editor-in-Chief *Michael Swaine*
Managing Editor *Vince Leone*
Editorial Assistant *Sara Noah Ruddy*
Technical Editor *Allen Holub*
Contributing Editors *Ray Duncan*, *Allen Holub*
Copy Editors *Laura Kenney*, *Rhoda Simmons*
Special Projects Editor *Frank DeRose*

Production

Production Manager *Bob Wynne*
Art Director *Shelley Rae Doeden*
Production Assistant *Alida Hinton*
Typesetter *Jean Aring*
Cover Design *Shelley Rae Doeden*

Circulation

Sub. Fulfillment Mgr. *Stephanie Barber*
Subscription Mgr. *Maureen Kaminski*
Book Marketing Mgr. *Jane Sharninghouse*
Single Copy Sales Mgr. *Stephanie Barber*
Circulation Assistant *Kathleen Shay*

Administration

Finance Manager *Sandra Dunie*
Business Manager *Betty Trickett*
Accounts Payable Supv. *Mayda Lopez-Quintana*
Accounts Payable Asst. *Denise Giannini*
Billing Coordinator *Laura Di Lazzaro*
Accountant *Marilyn Brown*
Adm. Coordinator *Kobi Morgan*

Advertising

Advertising Director
Shawn Horst *(415) 424-0600*

Advertising Sales
Walter Andrzejewski *(617) 567-8361*
Lisa Boudreau *(415) 424-0600*
Beth Dudas *(714) 643-9439*
Michele Beaty *(317) 875-8093*
Systems Manager *Ron Copeland*
Administrative Manager *Anna Kittleson*

M&T Publishing, Inc.

Chairman of the Board *Otmar Weber*
Director *C.F. von Quadt*
President and Publisher *Laird Foshay*

Dr. Dobb's Journal (USPS 307690) is published monthly by M&T Publishing, Inc., 2464 Embarcadero Way, Palo Alto, CA 94303, (415) 424-0600. Second class postage paid at Palo Alto and at additional entry points.

Address correction requested. Postmaster: Send Form 3579 to *Dr. Dobb's Journal*, P.O. Box 27809, San Diego, CA 92128. **ISSN 0278-6508**

Customer Service: For subscription problems call: outside CA 800-321-3333; within CA 619-485-9623 or 566-6947. For book, back issue, or disk order problems call 415-424-1474.

Subscription Rates: $29.97 per year within the United States. Foreign subscription rates: $56.97 for airmail, $46.97 for surface mail. Foreign subscriptions must be pre-paid in U.S. Dollars, drawn on a U.S. Bank.

Foreign Distributor: Worldwide Media Service, Inc., 386 Park Ave. South, New York, NY 10016, (212) 686-1520 TELEX: 620430 (WUI)

ABC Audited Paid Circulation

People's Computer Company

Dr. Dobb's Journal is published by M&T Publishing, Inc. under license from People's Computer Company, 2682 Bishop Dr., Suite 107, San Ramon, CA 94583, a non-profit, educational corporation.

LETTERS

Blum Signs Off

Dear *DDJ*,

Being a part of *DDJ* for practically three years has been an incredible experience. Even though I won't be writing a regular column anymore, I don't feel that I am losing a thing. The many new friendships that I now enjoy and the experiences that I shared will last a lifetime.

Of particular importance to me has been your feedback. The many letters I received, even when critical, were a delight to read and helped guide me in preparing material meaningful to you.

Even though I won't be a regular participant in future issues of *DDJ*, I have a number of CP/M projects under way that I hope will be accepted for publication in the future.

In closing for the last time, allow me to invite you to continue to visit my bulletin board system and to again thank each of you for looking in on the CP/M Exchange each month.

Bob Blum
5536 Colbert Trail
Norcross, GA 30092

Bob Blum's RCP/M system is available for your use 24 hours a day, 7 days a week. Reach it by dialing (404) 449-6588.

Dear *DDJ*,

I have been searching (with no success) for the address and subscription cost for *DTACK Grounded*. Would you or any of your readers have this information?

Thank you for your time.

Calvin Dodge
4490 Yukon Ct., #2A
Wheatridge, CO 80033

DTACK Grounded, *one of the best newsletters on programming and the industry, ceased publication with Issue 45 to allow its editor, Hal Hardenberg, to devote more time to software development. His company, Digital Acoustics, is healthy and is mailing subscription refunds to subscribers. Back issues are still available from Digital Acoustics, 1415 E. McFadden, Ste. F, Santa Ana, CA 92705.—ed.*

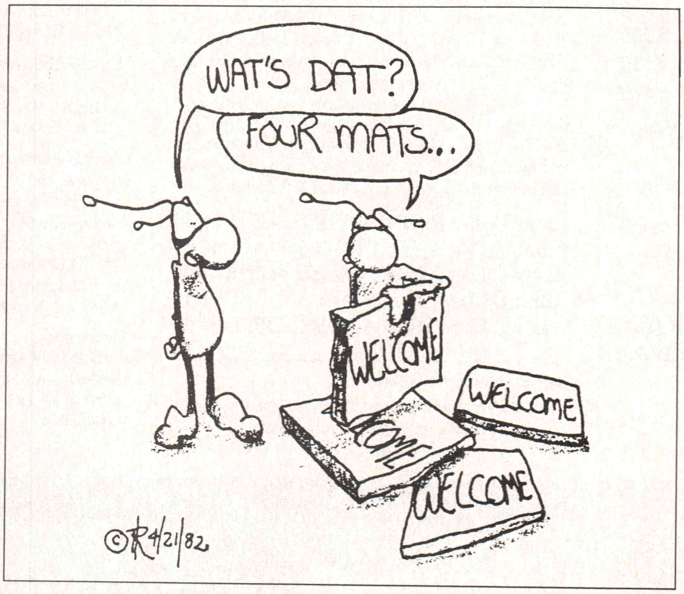
Editors

Dear *DDJ*,

Mark Edwards is to be congratulated for his review of editors in the November 1985 issue of *Dr. Dobb's*. Anyone who ventures into this highly personalized field takes his life in his hands, and to attempt a survey of ten editors is chutzpah indeed. To do it with taste, relative completeness, and fairness is no mean task.

As Mr. Edwards notes, EC is in a high degree of flux, but I would note that my version (probably later than his, given the lead time it requires for an article) can do backward searching and does not suffer the poor error handling he referred to. An important aspect of any software is that of support, and I would just like to comment that I have found the authors of EC superlative in that aspect.

A very nice feature of EC (among many others) is its maintenance of the DOS commands in a buffer. In all my editing I use APX Core, which offers multiple DOS partitions. EC did work with APX but lost the ability to maintain the DOS command buffer under it. I called Gene Brown, the principal author of EC, and informed him of this and also gave him the address of APX. Much to my surprise and delight, I received three (count them, 3) copies of the final version. One was the standard, and the other two were potential solutions to the APX compatibility problem. They all worked, and the solutions did so beautifully. How many companies will give such a response?



I have an additional comment with respect to the relevance of the compatibility of editors with such "enhancers" as SideKick. My experience has been that in the long run the side effects of these programs are less desirable than their offerings. Even more to the point, though, XyWrite, BRIEF, and EC (and probably others in the review also) are sufficiently internally complete that you really do not need such enhancers. BRIEF, with its powerful macro capability, allows you to create these enhancements for the most part, as does XyWrite for those it does not have; EC has most of them built-in, and the few it doesn't, you can create. Communications are accessible by any of these editors because you can exit to DOS, so you have the advantage of no overhead and no potential side effects. With BRIEF, I even wrote a perpetual calendar to mimic that aspect of SideKick.

Because people tend to use editors for short correspondence, a relatively important aspect is their "reformatting" speed as most editors do not (understandably) maintain word wrap for text insertion after a line has been entered. One of the problems I found with EDIX was its intolerably slow reformatting—in

(Continued from page 8)

this respect VEDIT and EC are lightning fast, and BRIEF (with its free reformat macro) is only slightly faster than EDIX. XyWrite of course does it like any respectable word processor.

I enjoyed reading "Wired Tales" in Dave Cortesi's column in the December issue and offer the following. I recently decided to upgrade my PC with a larger power supply (64 watts to 135 watts) and, being relatively frugal, ordered one from a mail-order house for $89. When it arrived I (foolishly) started to install it: removed my old power supply, put the new one in, and started to make the connections. To my dismay, it turned out that the connectors were poorly constructed and I could not get a good fit. I decided to return the supply for credit, not wanting to chance a repeat performance. It took me about $15 worth of phone calls to finally talk to someone who was authorized to issue an RA, and that same person informed me that there was a 10 percent restocking fee. Adding in the cost of the UPS shipment back, I wound up paying $30 for some futile labor in taking out and replacing my power supply, losing a nontrivial amount of time in the process.

Finally, I offer a poser. I recently purchased an uninterruptible power supply (300 watts) from Qubie. When it was delivered, I tested it in the usual way—I pulled the power plug—and it worked fine. Just to be doubly sure, I also opened the main circuit breaker in the house, and the same excellent performance repeated. A few days ago, we had our first real power outage while I was working at my PC, and much to my surprise, I lost all my work. When the power returned about one minute later, I immediately made the "pull the plug" test and it worked. I called Qubie's technical department, and no one could offer a solution, but the company did offer to send me a replacement (which I accepted). In the meantime I have been thinking about the possible causes of the loss of my work and believe I may understand why it happened, but I would love to hear from others.

Morton F. Kaplon
11 White Birch Dr.
Pomona, NY 10970

Dear *DDJ*,

Although I greatly enjoyed Mark Edwards' editor review (*DDJ*, November 1985), I would like to make a correction or two and point out a problem with this kind of review. I think Mr. Edwards did a terrific job given the difficulty of trying to learn about ten text editors, much less trying to comment intelligently about them all. I use Pmate (Version 3.37) on a daily basis, however, and I am familiar enough with it to have caught a couple of errors in the review.

The first (and smallest) is that Pmate does indeed allow you to undo the deletion of a single character. What it will not allow is the retrieval of a character you have backspaced away. On a PC, this is the difference between the Delete key and the ← (backspace) key. The difference can be annoying, but it is frequently more helpful than not once you are aware that Pmate works in this way.

A more serious error is in Mr. Edwards' Pmate macro to count braces. The macro he presents does indeed work, but it is terribly slow. Trying it on a sample C program out of one of my working directories (on a Compaq Plus), it ran for more than six minutes before I aborted it—unfinished. There is a much better Pmate macro for brace counting, which I present in Table 1, page 10. This brace-counting macro finished the same piece of code as above in less than six seconds (and is slowed slightly because it is constructed to work with a file of any size as opposed to one entirely in memory). This is a huge difference from the results reported in the review. Because of differences in the test files, it is impossible to say whether this performance is better or worse than BRIEF or EMACS, but it is certainly of the same order.

It is worth noting that the macro presented in Table 1 works in much the same manner as the BRIEF and EMACS macros presented by Mr. Edwards. In particular, it allows the editor itself to search for the characters to be counted rather than stepping through the file in a clumsy character-by-character fashion. I suspect that a properly rewritten macro for VEDIT PLUS would show the same kind of dramatic speed improvement. There might even be some hope for XTC if the proper macro formulation were used.

This in turn points out a serious difficulty in benchmarking editors. A user who is familiar with an editor will undoubtedly be able to generate a better benchmark than one who isn't. I would never have even considered writing a brace-counting program for Pmate in the way Mr. Edwards did. The way Pmate works guarantees that his macro is among the slowest possible for the task.

Based on my experience, I would tend to agree with Mr. Edwards' comments about the functionality and features of each editor he reviewed. Based on my findings for Pmate, though, I would be inclined to disregard the entire benchmark table except for items A, B, F, and maybe C.

Mr. Edwards has certainly proved that it is im-

```
;***
;***  Pmate brace counting macro program
;***
0v1          ;set variable 1 to 0
ua           ;go to top of file
[            ;begin loop
   e         ;suppress error messages
   us{       ;search for an open brace
   @t=0_     ;if none found, exit loop
   va1       ;else increment count of braces
]            ;end loop
ua           ;return to top of file
[            ;begin loop
   e         ;supress error messages
   us}       ;search for closing braces
   @t=0_     ;if none found, exit loop
   -1va1     ;else decrement count of braces
]            ;end loop
b2e          ;go to buffer 2
@1\          ;put the result in buffer 2
```

Table 1

Mark Williams

Now the biggest name in C compilers comes in a size everybody can afford.

Let's C.™

Introducing Mark Williams' $75 C compiler. Want to explore C programming for the first time? Or just on your own time? Now you can do it in a big way without spending that way. With Let's C.

This is no little beginner's model. Let's C is a powerful programming tool, packed with all the essentials of the famous Mark Williams C Programming System. The one chosen by Intel, DEC, Wang and thousands of professional programmers. The one that wins the benchmarks and the reviewers' praise:

"(This compiler) has the most professional feel of any package we tested..."—BYTE

"Of all the compilers reviewed, (it) would be my first choice for product development."—David W. Smith, PC WORLD

And now for more big news. Get our revolutionary csd C Source Debugger for just $75, too. You can breeze through debugging at the C source level ignoring clunky assembler code.

Affordable, powerful, debuggable. Mark Williams Let's C is the big name C compiler at a price you can handle. Get your hands on it now.

Mark Williams Let's C

- For the IBM-PC and MS-DOS
- Fast compact code plus register variables
- Full Kernighan & Ritchie C and extensions
- Full UNIX™ compatibility and complete libraries
- Small memory model
- Many powerful utilities including linker, assembler, archiver, cc one-step compiling, egrep, pr, tail, wc
- MicroEMACS full screen editor with source
- Supported by dozens of third party libraries
- Upgradeable to C Programming System for large scale applications development

Let's C Benchmark Done on an IBM-PC/XT, no 8087. Program: Floating Point from BYTE, August, 1983.

	Exec Time in Seconds
Let's C	134.20
MS 3.0	347.45

Use this coupon or charge by calling toll-free: 1-800-MWC-1700. In Ill. call 312-472-6659.

Mark Williams Let's C

$75

Please send me:

_____ copies of Let's C and _____ copies of csd (C Source Debugger) at $75 each. (Ill. residents add 7% sales tax.)

☐ Check ☐ Money Order ☐ Visa, MasterCard or American Express

Name_____________________

Address_____________________

City_____________ State_____ Zip_______

Card #_____________ Exp. Date_________

Signature_____________________

Mark Williams Company
1430 West Wrightwood
Chicago, Illinois 60614

Circle no. 102 on reader service card.

(Continued from page 10)

possible for a single person to learn enough about ten different editors to be able to compare each at its best.

Brad Chase
P.O. Box 705
Exeter, NH 03883

Modula-2

Dear *DDJ*,
While perusing your otherwise excellent November issue, I felt a sudden stab of pain. Looking closer, I saw the source of my discomfort was "Bit Manipulation in Modula-2." I feel an overwhelming need to comment.

"I knew from the outset that it was impossible to match C's simplicity." This tells me that the author is primarily a C hacker. Modula-2 is perfectly capable of matching this "simplicity," and in exactly the same manner. (See Table 2, below.)

The other operations are left as exercises for the reader.

Lawrence C. Smith
51 Lake St.
Nashua, NH 03060

```
(* assume ch:=CHAR(255) *)                    (* clearing bits *)
(* clear bit 7 *);
ch:=CHAR(BOOLEAN(ch) & BOOLEAN(07FH));
(* result is ch=CHAR(127) *)
(* assume ch:=CHAR(15) *)                      (* setting bits *)
(* set bit 5 *)
ch:=CHAR(BOOLEAN(ch) OR BOOLEAN(10H));
(* result is ch=CHAR(63) *)
```

Table 2

C Compilers

Dear *DDJ*,
I love *Dr. Dobb's* and have subscribed for years. I notice a trend away from assembly, CP/M, 8-bit, and the less wealthy user, though. In particular, your issue on C language was a disappointment. The following are my reasons, along with some suggestions. I understand your problems in appealing to a divergent group, however, and appreciate your willingness to listen to all our suggestions.

1. The main article on C comparisons told me more than I really cared to know about the languages. I would have trusted the writers if they had summarized their results. An article this long and detailed should have a summary at the beginning.

2. The article did not state which of the languages examined were available on either S-100 machines, on CP/M, or on 8-bit machines.

3. I did not see mention of several of the Cs I use—BDS C, Small-C, or Tiny C or any mention of Lifeline's products—and that would have been important to me. I am sure other Cs could have been found as well—so how much longer would it have taken to wait a bit and do a truly definitive review?

Frederic Schlamp
2205 Meadowview Rd.
Sacramento, CA 95832

The comparative review of C compilers in our August 1985 issue was strictly dedicated to MS DOS-compatible compilers. We will review C compilers for other environments, including CP/M, in 1986.—ed.

Columns

Dear *DDJ*,
In your July 1985 issue the 16-Bit Software Toolbox column began with the line "One of the most novel features added in Version 2 of MS DOS is the concept of 'installable device drivers.'" I would like to say that this concept may be new and novel for Microsoft and MS DOS, but it is certainly not a new and novel concept for other operating systems available for microprocessors.

The OS-9 operating system has had the concept of user-installable device drivers since its initial 6809 Level 1 release in 1978. In fact, OS-9 is totally modular in nature and allows the user to add new device descriptors, device drivers, and new file managers if required. In addition to supporting installable drivers, OS-9 has included the "novel" MS DOS concepts of hierarchical director structures and pipes. OS-9 also gives full support to I/O redirection, multiprocessing, and multitasking—concepts much more akin to Unix.

OS-9 may not be as well known as MS DOS is, but it does have a large and rapidly growing following in the 6809 and 68XXX world today. MS DOS has added nothing novel to its OS; it is adding features that are expected and required for an OS in today's world. These features have been around in OS-9 and OS-9/68000 for some time.

Tim Harris
Microware Systems Corp.
1866 N.W. 114th St.
Des Moines, IA 50322

Dear *DDJ*,
"It isn't what you don't know that hurts you, it's what you know that ain't so." In the September Of Interest column, the author states that APL on the IBM PC requires an 8087. This is only true for IBM's own APL, not for the other five (STSC, Sharp, Portable Software, Watcom, and NIAL Systems). Most of them run on the PCjr.

Edward M. Cherlin
6611 Linville Dr.
Weed, CA 96094

Dear *DDJ*,
I was interested to read the portion of your Dr. Dobb's Clinic (October 1985) regarding the intricacies of manipulating path names. Some time ago, we at POLYTRON also had the pleasure of figuring out how to do exactly what you discussed. Our solution, which produced some functions that are more general purpose, might interest your readers. The primary components of our solution are:

1. A function to determine if a given path name represents a directory—that is, it is a drive ID or an actual director but not a normal file:

```
int is_dir(name)
char *name;
```

2. A function to determine if a given file name contains wildcard characters:

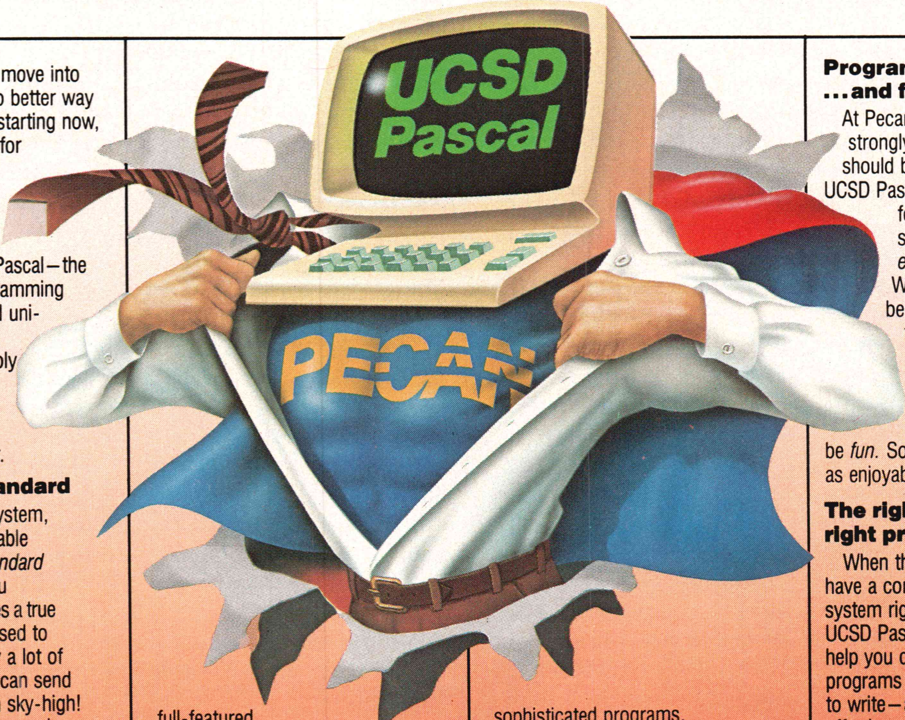
```
int is_wild(name)
char *name;
```

3. A general-purpose function for generating a new file name given an existing file name, an optional new extension, and an optional new path:

```
char *
bld_fnam(pathp,
   namep, extp)
char *pathp;
```

Turbo, who?

Do you have to give up power and advanced potential to get ease of use and affordability? Not anymore. Because now, you can have *UCSD Pascal* for only $79.95!



If you're making your move into programming, there's no better way to go than Pascal. And starting now, you *don't* have to settle for a stripped-down version of Pascal in order to get a price that's right. Instead, you can choose UCSD Pascal—the recognized Pascal programming standard in colleges and universities throughout the country—at the incredibly low introductory price of $79.95 for your PC-DOS, MS-DOS, or other popular computer.

Start with the standard

With an *entry-level* system, you spend a lot of valuable time learning a *non-standard* form of Pascal. And you don't get all the capabilities a true Pascal system is supposed to deliver—unless you buy a lot of add-on utilities—which can send the cost of your system sky-high! Worst of all, when you're ready to tackle anything more than short, simple programs—you have no choice but to move up to a more sophisticated system (like UCSD Pascal). And at that point, you also have to *relearn* standard Pascal.

UCSD Pascal has everything you need

With UCSD Pascal, you get a full-featured, professional programming tool that's being used right now in the development of major scientific and business applications. The system comes with an outstanding text editor, a complete on-line tutorial, 8087 math coprocessor support and BCD (decimal arithmetic) *included* in the package at no extra cost. In fact, UCSD Pascal contains virtually everything you need—as standard equipment—for developing the simplest to the most sophisticated programs.

UCSD Pascal is available for MS-DOS, PC-DOS, UNIX, VMS, MSX and many other operating systems. You can use UCSD Pascal to write programs of any size on virtually any computer, and port them to any other computer. And if speed is what you're after, the latest native code version of UCSD Pascal actually benchmarks favorably with Turbo Pascal® in execution time!

Programming that's easy ...and fun!

At Pecan Software Systems, we strongly believe programming should be as *easy* as possible. UCSD Pascal was originally designed for teaching programming skills, so it's extremely *easy* to learn and to use. With UCSD Pascal, you'll be developing programs right from the start that are *easy* to write, *easy* to understand, and *easy* to maintain. We also believe that programming should be *fun*. So we've made UCSD Pascal as enjoyable to use as it is powerful.

The right tool at the right price

When the fun gets serious, you'll have a comprehensive programming system right at your fingertips with UCSD Pascal—a system that will help you develop those big-league programs you may eventually want to write—at a price you can readily afford.

Put UCSD Pascal programming power on your PC now for only $79.95! **Order by mail today or phone now 1-800-63-PECAN.** UCSD Pascal—*the* original standard of Pascal programming excellence. The new leader in Pascal price/performance.

PECAN™

The UCSD Pascal Company
Pecan Software Systems, Inc.
1410 39th Street, Brooklyn, NY 11218
718-851-3100

UCSD Pascal™

UCSD Pascal
Not copy protected
60-day money-back guarantee

UCSD Pascal $79.95 (for PC-DOS, MS-DOS, AMIGA, APPLE, ATARI 520, MACINTOSH, RAINBOW, TANDY, as well as most popular 8/16/32-bit systems).
Price includes 8087 support and BCD.

Please send me ____ copies of UCSD Pascal for my ____________ (Name and model of computer) which runs under ____________ (Operating system).

My disk size is 3½" ____ 5¼" ____ 8" ____.

Total amount (NYS add appropriate tax) ____________.

Payment by ☐ VISA ☐ MC ☐ US Bank Check ☐ Bank Draft

Card Number: ____________ Credit Card Expiration Date: ____/____

Mail to: Pecan Software Systems, Inc.
1410 39th Street
Brooklyn, NY 11218
ITT Telex No. 494 8910
CompuServe Code 76703, 500

CREDIT CARD ORDERS
CALL TOLL-FREE
1-800-63-PECAN
(NYS) 1-800-45-PECAN

Name ____________

Shipping Address ____________

City ____________ State ______ Zip ______

Telephone ____________

Call or write for UNIX, VAX or other UCSD Pascal versions—and ask about our powerful Pascal add-ons, too. SCHOOLS: Contact us for our special educational discounts!

Call toll-free or enclose a check with this coupon to place an order. Please add $2.50 for shipping within the US. Foreign orders add $10 and make payment by bank draft payable in US dollars on US bank. New York State residents add appropriate sales tax.

DD

AMIGA is a trademark of Commodore Electronics LTD.
APPLE & MACINTOSH are trademarks of Apple Computer.
ATARI 520 is a trademark of Atari Corporation.
RAINBOW is a trademark of Digital Equipment Corporation.
TANDY is a trademark of Radio Shack.
Turbo Pascal is a registered trademark of Borland International

Circle **no. 200** on reader service card.

(Continued from page 12)

```
        /* the new path */
char *namep;
        /* the original name */
char *extp;
        /* the new extension */
```

This returns a pointer to allocated memory containing the newly constructed file name. The original name may or may not have a path or extension. If either *pathp* or *extp* point to a null string, the respective component of the original name is omitted from the new name. If either *pathp* or *extp* is a null pointer, the respective component of the original is used in the new name.

4. A wildcard file name expansion function with optional ability to prepend the path of the wildcard name to each matching name is shown in Table 3 (below). This returns a pointer to a linked list of names matching the wildcard name. Each node of the list is contained in separately allocated memory and is of sufficient length to contain the matching name beginning at *name[0]*.

5. Functions to extract each of the three components of a file name—the path, the root, and the extension are shown in Table 4 (below). These functions each return a pointer to allocated memory containing the respective component of a file name.

Using these functions, you can do just about anything you wish with file names. In fact, they are used in nearly all our products and continue to be used in new development work.

These functions, along with a host of others, are available in the POLYTRON C Library 1. This package contains routines that function under DOS 1.x, DOS 2.x and later, and both/either. The functions are provided in linkable form (libraries), as well as in full source form (C and assembler) for the IBM PC/XT/AT and compatibles.

Donald K. Kinzer
POLYTRON Corp.
P.O. Box 787
Hillsboro, OR 97123

```
/* element of a linked list of matching names */
struct file_list
{
    struct file_list *next; /* ptr to next node */
    char name[1];       /* a file name */
};
struct file_list *
expand(wildname, add_path)
char *wildname;         /* the name to match */
int add_path;           /* non_zero if path of wildname should be
                           prepended to matching names */
```

Table 3

```
char *
path_of(name)
char *name;         /* the name from which to extract the path */
char *
root_of(name)
char *name;         /* the name from which to extract the root */
char *
ext_of(name)
char *name;         /* the name from which to extract the
                       extension */
```

Table 4

Dear *DDJ*,

Here is a report on a software company for the benefit of prospective buyers. I purchased a cross-assembler from 2500AD in November 1984. What they sent me was a nonfunctioning program. Soon after receiving the program, I sent two letters, made two phone calls, and finally spoke to someone who said the company would send me a good copy when it had fixed the program. It has been a year, the firm still has my $200, and I have not yet received a functional copy of the program. Even JRT Pascal was functional, and look at the price difference!

For the record, here are the defects that I have found (so far):

1. The assembler and the linker are incompatible. The linker makes the wrong assumptions about the relative order of least/most significant bytes, which makes it impossible to link object modules. The only way to get an executable file is to put all your code into one module and use absolute addressing (with an ORG statement).

2. The assembler gives the wrong machine code for several of the opcodes. This makes for difficult debugging.

3. Some of the pseudo-ops listed in the manual don't exist in the actual assembler.

4. Some of the pseudo-ops that do exist do not work (at least one).

5. The assembler has no provision for allowing the programmer to specify the short forms of the relative jump and call instructions.

6. The symbol table that is printed at the end of the listing usually contains sections of garbled mess.

7. The printed symbol table lists all intermediate values of labels that are redefined many times with the DEFL statement.

8. Some errors in the source code cause the assembler to crash with no error message.

9. The linker crashes with a nonsense error message under some conditions that seem to have something to do with certain exact lengths of files.

Neil R. Koozer
Kellogg Star Rt.
Box 125
Oakland, OR 97462

In the September issue, we ran a comparative review of PC TEX and MicroTEX. At that time only PC TEX included INITEX, but we stated that MicroTEX was also scheduled to include INITEX in an August update. As of the time this issue went to press, MicroTEX still does not include full INITEX capabilities. We have been informed by Addison-Wesley that a new version of MicroTEX, including INITEX, will not be available until January 1, 1986. Check with Addison-Wesley before placing an order. Both Addison-Wesley and Personal TEX are already distributing the Textset laser printer and screen preview driver.—ed.

DDJ

Circle **no. 77** on reader service card.

Inefficient C

This column is adapted from DTACK Grounded, The Journal of Simple 68000/32801 Systems, *Issue 42.*

It has been apparent to me for two or three years now that complex programs (as opposed to the famous but simplistic "hello world" type) written in C consistently run a great deal slower than the same complex programs written in assembly. Examples of this rule can readily be found in the personal computer mass marketplace.

It is true that, in the last year or two, many intelligent and experienced programmers have asserted that C has little or no high-level language (HLL) overhead.

There is an apparent conflict here.

Consider the complex problem of writing a BASIC interpreter. You can break this problem down into a large number of simple problems that can then be solved in assembly or, say, C. The speed with which each of the simple problems can be solved depends on how good a match can be made between the problem and the available control constructs, data types, and so on. In some cases the match with the constructs and data types available in

by Hal Hardenberg

C is as good as the best that can be devised using assembly, and you have zero HLL overhead. In fact, many simple problems can be solved as quickly (or nearly so) in C as in assembly.

So, the C supporters have no shortage of real-world examples of simple problems that can be solved as quickly (or nearly so) in C as in assembly, which leads them to claim that C has no high-level overhead.

Some problems, though, can be solved easily and quickly in assembly and are real bears in C, given the restricted range of operations, constructs, and data types available to C programmers. The large number of simple problems that comprise a BASIC interpreter will include some C-easy problems, some C-not-so-easy problems, and some C-bears. Therefore, any full-function BASIC interpreter written in C will always be a great deal slower than the same interpreter written in assembly.

You need not look for complex ways to analyze the HLL overhead of C (or any high-level language). The fact is, all high-level languages greatly restrict the range of operations, control constructs, and data types available, compared to assembly language. Thus, HLL programmers have a limited number of tools available for solving the large number of simple problems that comprise any solvable complex problem.

Obviously, the programmer with the most complete set of tools can always produce the fastest code. Real-world evidence such as Microsoft's MBASIC (written in assembly) vs. DRI's Personal BASIC (written in C) confirms this rule.

Why, then, do intelligent, experienced programmers claim that C has little or no HLL overhead when abundant real-world evidence contradicts such an assertion?

Well, suppose you have solved a complex problem using C. Then, keeping the 90-10 rule in mind, you go looking for ways to speed up the software. Do you rewrite the program from scratch in assembly, using the greater variety of available operations, data structures, and control constructs? No, you continue to use the data structures and algorithms that you developed in C. That is, you implicitly restrict yourself to the smaller toolkit that is already used by C! Not surprisingly, the resultant program is not much faster than the original. Hence you announce, "Hey, guys, I tried optimizing the problem using assembly and got little or no improvement over the original C. Obviously C has little or no high-level overhead."

Obviously.

HLL inefficiency is acceptable in many environments. Where HLL inefficiency is not acceptable is in the personal computer mass marketplace wherever an efficient alternative is available. DRI's Personal BASIC is essentially a dead issue, being highly uncompetitive with Microsoft's assembly-based BASICs in terms of speed. In turn, the marketplace is widely avoiding Microsoft's C-written FORTRAN compiler.

The individuals who make up the mass marketplace and who vote with their wallets don't care how hard it was to produce a program or how long it took. They don't care about source documentation or maintainability because they never get to see the source documentation and somebody else maintains the code. They vote for those programs that work both swiftly and well (Lotus 1-2-3, original WordStar, MBASIC, or GW-BASIC) rather than those that run well but slowly (Context MBA, WordStar 2000, or DRI Personal BASIC).

The folks who assert that C has little or no HLL overhead are either talking about simple problems for which C is in fact well suited, or they are observing that, if algorithms written in C are duplicated in assembly, then the assembly versions have little advantage. To reason that C therefore has little HLL overhead is faulty logic.

Other things being equal, the more complete toolbox is always going to win provided the workman knows how to use those tools. That's why good, experienced assembly programmers cost more than HLL coders.

DDJ

DDJ ONLINE

DDJ Goes Online

On January 1, 1986, the Electronic Edition of *Dr. Dobb's Journal* will appear on CompuServe. Through the Electronic Edition *DDJ* will offer the following:

- We will make available in our data libraries all the listings that appear in the articles and columns of *Dr. Dobb's Journal* every month. We will also offer selections from some of our more popular back issues and run articles and listings that have not appeared in the magazine.
- We will maintain a display area where CompuServe users can read abstracts of the material in the current issue. The display area will also contain general magazine information, such as our editorial calendar, writers' guidelines, and information about advertising rates.
- We will compile comprehensive lists of the commercial software development tools available to microcomputer programmers and feature comparative reviews of products in selected categories. We will also compile selected bibliographies and give capsule reviews of newly released books.
- We will provide a messaging facility with columnists and other SIG members.
- We will stage regular online teleconferences with authors, columnists, and other distinguished guests.
- We will take subscriptions for *Dr. Dobb's Journal*, the magazine. We will also provide a way for readers to register circulation complaints.
- We will provide complete information about other *DDJ* publications, such as back issues, bound volumes, indexes, *Dr. Dobb's* Books, and *Dr. Dobb's* Software. We will also take orders for these items.

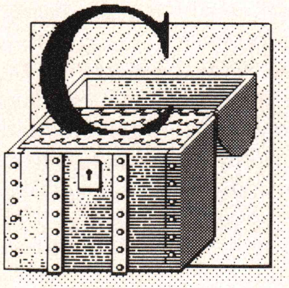
Please watch this department for further announcements about conference schedules or the availability of listings from back issues. Also, if you have a request for a listing from a back issue, drop us a note or just leave a message in the *DDJ* Electronic Edition on CompuServe. If we get enough requests for a specific listing, we will try to make it available.

Some of these services may not be available January 1. Most will be free, but some will involve a small charge.

You access the *DDJ* Electronic Edition by typing go *DDJ* at any CompuServe system prompt. We hope you will drop by and have a look. See you soon!

DDJ

A Unix-like Shell for MS DOS



Last month we presented a few support routines, part of a Unix-like shell for MS DOS. This month we're going to continue with the shell itself, describing how it works on a high level. Next month we'll look at it at a lower, functional level. The code itself, because it's so long, will be spread over the next three months.

The shell described here includes functions of the Unix C shell that I use most often, such as aliases, history, and command-line wildcard expansion (more on all this in a moment). It has batch-file capability and will permit nested batch-file execution (unlike MS DOS, which lets you chain from one batch file to another but won't let you return to the original batch file).

It supports a 2,048-byte command line with interactive editing. The long command line is passed to an executed program through an environment variable.

Using the Shell

Commands are entered from the command line, just as in DOS. (Note that \ is a special character to the shell, so use slash (/) or \ \ to separate directory names.) DOS wildcard characters (* and ?) are expanded before a command is executed. So if you say *echo *.c*, the **.c* will be expanded to the names of all files having a .c extension before *echo* is invoked. Expanded names are sorted. Several semicolon-delimited commands may be executed from a single command line. For example:

```
cd foo ; pwd ; ls
```

by Allen Holub

changes the current directory to *foo*, prints the full path name, and then prints a list of the files in the current directory.

Command-line editing (as described last month) is supported. To summarize:

Cursors—moves the cursor.
Home—gets to the beginning of the line.
End—gets to the end of the line.
Ctrl-right arrow and Ctrl-left arrow—get to the next and the previous word, respectively.
^H—is a destructive backspace.
Del—deletes the character under the cursor. (Typed characters will be inserted at the current cursor location, moving all characters to the left of the cursor over one notch.)
^X—deletes the entire line.
Esc—does the same and aborts.
Return—causes the commands to be executed.

There are several built-in commands:

alias—creates, modifies, or prints aliases (see below). There are two syntaxes:

```
alias
alias name <val>
```

The first prints all currently defined aliases, and the second creates an alias for *name* with the indicated value. *<Val>* may be anything on the command line, but you have to escape (precede with a \) any character that's special to the shell (or surround *<val>* with double quotes).

cd—changes a directory or disk:
cd foo—changes to the subdirectory *foo*.
cd..—changes to the parent directory.
cd a:—changes to the current directory on the a: drive.
cd a:/foo—changes to the */foo* directory on the a: drive.

Cd must be used to change disks, although you can *alias a:* to *cd a:* if you like. The shell checks to see if a disk is in the indicated drive before the drive is logged on.

exit—terminates the shell. Either *exit* or *logout* must be used to leave the outermost shell. Subshells can be terminated with a *^C.*

history—prints the history list (see below).
logout—like *exit*, terminates the shell; however, the file /logout.bat is executed before exiting.
pwd—prints out the current working directory (same as *chdir* with no arguments under DOS).
rem—does nothing. May take arguments (which will be ignored). This command is here only for DOS compatibility. The preferable method for commenting a batch file is to start comment lines with a # in the far left column. Note that *rem* is interpreted as if it were a command; that is, the line is put into the history list. Moreover, if a line starts with a *rem* but has a semicolon on it as well, text up to the semicolon will be ignored but the semicolon will be treated normally, and any text following the semicolon will be treated as a second command and executed.
set—creates, modifies, or prints a shell variable (see below). Its syntax is:

```
set
set name [=] [value]
```

The first form prints all current shell variables, and the second creates or modifies an existing variable. Both the equals sign and the value fields are optional. If

you omit the value, the alias will expand to a null string.

setenv—changes or creates an environment variable. Its syntax is:

setenv name [=] [value]

Both the equals sign and the value field are optional. An example:

setenv PROMPT $p>

sets the prompt to the current directory name followed by >. This command is very similar to the DOS set command. However, *setenv* with no arguments doesn't print the environment. Also, the equals sign is optional.

shift—shifts all the *$<num>* arguments in a batch file left one notch. For example, if a batch file (foo.bat) consisting of:

```
echo $0 $1 $2
shift
echo $0 $1 $2
shift
echo $0 $1 $2
```

is executed with the line:

foo first second

the following output will be sent to the screen:

```
echo first second
first second
second
```

(*$0* expands to the file name.)

unalias—removes an alias. The syntax is *unalias name*.

unset—removes a shell variable. The syntax is *unset name*. The default variables (*verbose, echo, arg*) can't be removed.

History

The history functions let you examine, edit, and reexecute previous commands. The 32 most recently typed commands are remembered in a history list, and each command has a history number associated with it. Once the history list is full, the oldest command is discarded every time a new command is added to the list. The easiest way to see how this mechanism works is with an example. Assume that you've typed the following five commands:

```
vi prog.c
cc -c prog.c >err
vi another.c
cc -c another.c >err
link another+prog,prog,prog,c.
    lib+/lib/tools.lib
```

As you type the commands, they're entered into the history list, which can be examined by typing *history*. The following will be printed:

```
1: vi prog.c
2: cc -c prog.c >err
3: vi another.c
4: cc -c another.c >err
5: link another+prog,,,c.lib+/lib/
     tools.lib
6: history
```

Note that *history* itself is also added to the history list. The numbers are the history numbers associated with each command. A command can be executed again by typing *!#*, where *#* is the history number. Typing *!5* causes the link command to be executed again. *!2* will reexecute the first

C CHEST:
(Continued from page 19)

cc command. You can also type *!<pat>*. In this case, the history list will be searched backward for a command starting with *<pat>*. So *!l* or *!link* would also redo the link. *!cc* would be the same as *!4* (because the first matching command is used). *!!* will repeat the last command you typed. In the above example, *!!*, *!6*, and *!h* will all do the same thing. Commands are added to the history list every time they're executed, even if a command is an expansion of a history request.

Several additional non-Unix history commands are supported. *!>file* will write the current history list to the specified file. If no file is given, */histlist* is used. The complementary command is *!<file*, which adds the commands in *file* to the current history list. The commands aren't executed, they're just added to the list. Again, if *file* isn't specified, */histlist* is used. Neither *!>* nor *!<* will show up in the history list (they won't use up a history number, either).

^ may replace *!*; that is, the commands *^^*, *^#*, and *^<pat>* may be used in place of *!!*, *!#*, and *!<pat>*. The *^* commands work just like the *!* commands except that the line is brought into an edit buffer that you can then manipulate in the normal way (with the cursors, etc.) before executing it. Unlike DOS, the command line is visible while you're editing it. Note that Esc will abort out of edit mode without executing the edited command line.

Environments, Shell Variables, and the Set Command

Shell variables are macros. They let you associate a body of text with a name, and when that name is used, the corresponding text is substituted. Shell variables are created with the *set* command and deleted with the *unset* command. They work something like Unix and DOS environments except that they can't be passed to a child process. Once a shell variable is created, it can be used anywhere in a command. For example, you can define a shell variable to represent a long directory spec with:

```
set HOME /usr/allen/src/shell
```

You can then use it on the command line:

```
cd $HOME
```

Cd $HOME will be expanded to *cd /usr/allen/src/shell* before the shell executes the line. Note that *$* must precede all uses of shell variable names but must not be in the definitions; *%* may be used instead of *$* if you prefer. There are several predefined shell variables (which can't be modified with the *set* command). These are:

$<num>—an argument to a batch file ($0 is the file name, $1 the first argument, etc.).
$*—expands to all *$<num>* variables concatenated together.
$p—expands to the current path name.
$!—expands to the current history number.
$s—expands to the current shell level.

Still using dBASE†?

If you're still paying for multiple copies of dBASE or RunTime† at hundreds of dollars a pop, you can stop.

And use a WordTech compiler instead.

With a WordTech compiler, you only need a single copy of dBASE II† or dBASE III† for every programmer, not every user.

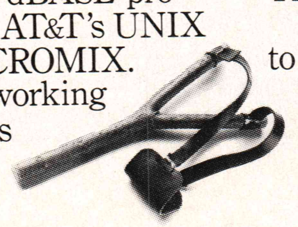
Then you can distribute as many copies of your compiled (and protected) programs as you want with no site licenses, no runtime fees and no strings attached.

Our MS-DOS compilers typically run your dBASE applications 3 to 10 times faster and use just 128K of free memory on any MS-DOS computer, not only the IBM PC.

You don't pay any penalties because WordTech compilers use the same language and syntax as dBASE and the same index, memory and data files (up to 10 data files open simultaneously, each with up to 7 indexes).

And you don't pay the usual high price because WordTech compilers are just $750. Once.

With a WordTech compiler, you can also run your programs on multi-user and networked systems. Our multi-user compiler runs your existing dBASE programs under AT&T's UNIX System V or CROMIX. And our networking compiler runs your present code on



DOS 3.1 local area networks (LAN's). The compiler will take care of file and record locking for you, or you can take full control using dBASE III networking commands.

We can help you extend your applications as well as your budgets. You can do windows with dBFrame™, business graphics with dBChart™ and even replace dBASE itself for just $169 with dB-XL™, an interpreter with an extended superset of the dBASE III language.

And we back everything with free support, one year of maintenance and a money-back guarantee.

For details and the name of your nearest dealer, contact WordTech Systems, Inc., P.O. Box 1747, Orinda, CA 94563. (415) 254-0900. TELEX 503599.

You owe it to your wallet to call now.

WORDTECH SYSTEMS

Circle **no. 208** on reader service card.

C CHEST:

(Continued from page 20)

The top-level shell is 0. If you create a shell within a shell, the second one will be at level 1. All batch files are executed in their own shells.

There are three other shell variables that can be modified with *set* but can't be expanded with *$name*. These are *echo*, *verbose*, and *cmd*. If *echo* is set (with either a *set echo* or a *set echo = 1*), then commands will be echoed to standard output just before they're executed. The default is *echo off* (unlike MS DOS), so you must set *echo* inside a batch file if you want to see what the batch file is doing. The echoed line will show all macro substitutions and all wildcard expansions; however, it will be truncated to the 127 characters permitted by DOS. You can use this feature to see if the command line has been truncated when you're invoking a program that doesn't know about the CMDLINE environment (see below).

Verbose shows the input as the shell receives it (before it's interpreted). If *cmd* is set (the default), then an environment variable called CMDLINE will be created every time a file is executed. CMDLINE holds the entire 2,048-byte command line (which can't be passed via DOS). If *cmd* is cleared, then CMDLINE is still created but will have no contents. *Echo*, *verbose*, and *cmd* can be cleared with *set <name> = 0*. Note that setting *echo* or *verbose* has the same effect as specifying −x or −v on the command line used to invoke the shell.

Environment variables (or strings) are similar to shell variables except that they can be passed to a child process (a pointer to them is included in a child's PSP). Many compilers (at least the Aztec, Lattice, and Microsoft) have a *getenv(name)* function in their libraries that returns a pointer to the environment string corresponding to *name*. If you need to write your own *getenv( )*, a good description of the PSP can be found in *The Peter Norton Programmer's Guide to the Norton IBM PC* (Bellevue, Wash.: Microsoft Press, 1985), pp. 260f.

Environment variables can be set from within the shell with the *setenv* command. Unlike DOS, they can be used on the command line just like a shell variable (precede the name with *$* or *%*).

Special Characters

* and ?—have the same significance as in MS DOS. They are expanded by the shell to matching file names. Expanded names are sorted. For example, *echo *.c *.obj* would print a list of all the .c files in the current directory (sorted) followed by all the .obj files (also sorted).

;—is used to separate multiple commands on a single command line. *Cd foo;pwd* would change the current directory to *foo* and then execute the *pwd* command.

\—is used to take away the significance of a special character. For example, * can be used to pass an asterisk to *grep* (the * won't be expanded). \ ; can be used to define a multiple command alias (see below). \\ evaluates to a backslash. The \ will be stripped from the line before the line is passed to the child process.

quotes—text surrounded by double

(") or single (') quotes won't be modified (wildcard characters aren't expanded, semicolons aren't interpreted as command delimiters, etc.). The quotes aren't removed unless the $-q$ argument is given on the command line. Unlike Unix, there's no distinction made between single and double quotes.

#—when found in the far left column, signifies a comment. The remainder of the line is ignored, and the line isn't put into the history list.

Aliases

Aliases are another sort of shell-maintained macro. Unlike shell variables, a $ is not needed to expand the name; rather the alias is expanded if its name is found as either the first word on a line or the first word following a semicolon on a multiple-command line. Aliases have two uses: They can be used to change the name of a command and they can be used in place of batch files. Aliases are created with the *alias* command. Some examples: If you have a program called *ls* that prints the current directory but you also want to be able to type *dir* and get a directory listing, you can define an alias for *dir* as follows:

```
alias dir ls
```

Thereafter, when the shell finds *dir* as the first word on a line, it will substitute the string *ls* for the string *dir*, and the program *ls* will be executed. Only the first word of a command is modified so *dir foo bar rat* will be changed to *ls foo bar rat*. Aliases must be either the first word on a line or the first word following the semicolon when there are several commands on one line.

Aliases can also be used in place of batch files, provided that no arguments need to be expanded. Because aliases are memory resident, they will execute much faster than a batch file. Similarly, an alias doesn't execute under its own shell, as does a batch file, so much less core is needed to execute an alias than is needed to execute a batch file. A simple alias that is similar to a batch file is:

```
alias shell cd /src/util/shell
```

Now you can type the single word *shell*, which the shell will expand to *cd /src/util/shell* and then execute to move to the indicated directory; however, you can't expand arguments. The portion of the command line that follows the alias name will be concatenated to the end of the expanded alias. Let's look at an example: Polymake lets you specify a default rules file on the command line with a −B option, but you often have to make a target name as well. By defining the alias *m* with:

```
alias m make -B /lib/
                    builtins.mak
```

you can then type *m foo.exe* and the shell will expand it to:

```
make -B /lib/builtins.mak
                    foo.exe
```

Because you can have multiple semicolon-delimited commands on one line, you can define an alias that will expand to several commands. For example:

C CHEST:
(*Continued from page 23*)

```
alias m rm err \; make -B /lib/
                       builtins.mak
```

will expand to

```
rm err ; make -B /lib/
                       builtins.mak
```

thereby both removing the file *err* and executing *make*. Note that you must escape the semicolon (with a \) in the alias definition so that the shell won't intercept it. (You could also surround the definition with quotes.)

Aliases may use shell variables. Two such aliases are:

```
alias here set here = \$p
alias there cd \$here
```

Here will set the shell variable *here* to the current directory. *There* will put you in the directory remembered with a previous *here* invocation. Note that *$* has to be escaped to prevent the shell from expanding it when the alias is defined.

A caveat about aliases: Aliases defined in terms of other aliases won't work. For example:

```
alias foo echo foo
alias bar echo bar
alias foobar foo;bar
```

won't work (*foobar* will expand to *foo; echo bar*). However, the command line *foo;bar* will work. Also note that commands are added to the history list *before* aliases are expanded.

Environments and Files

The default command line prompt is *[$s,$!]* (which prints the current shell level followed by the current history number). You can specify a different prompt with the PROMPT environment variable (use either the DOS *set* or the shell's *setenv* command). Any ASCII character may be used, and any of the *$* arguments will be expanded before the prompt is printed. For example:

```
setenv PROMPT $p->
```

will change the prompt to the current directory name followed by - and >. The prompt can be changed at any time (it doesn't have to be set when the shell boots). The SWITCHAR environment variable tells the shell what character signifies a command-line switch when it's the first character in a command-line argument (the default is -). Because environments are inherited by the child process, SWITCHAR is also available to a program if it chooses to use it.

The CMDLINE environment holds the complete, 2,048-byte command line (which can't be passed via DOS). It is changed every time the shell executes a command. Note that the command line (truncated to 127 characters) is also passed to a child process in the normal way (via the DOS command line buffer at offset 0x80 from the child's initial code segment). When the shell spawns a subshell to execute a batch file, it uses CMDLINE.

The SHLEV environment is set to the current shell level. The outermost shell is at level 0. All shells created from within another shell (including those used to execute batch files) will have higher numbers, de-

pending on the level of nesting.

Several files are used by the shell. If it exists, the file */shrc.bat* is executed (in a manner analogous to *autoexec.bat*) every time a shell is created. Because batch files are executed in their own shells, */shrc.bat* will be executed every time a batch file is executed. A second file */login.bat* is executed only once, when the lowest-level (level 0) interactive shell is created. *Shrc.bat* is executed before *login.bat*, and both files are executed before the environment is examined. My own *login* file is shown in Table 1, at right. My *logout* file is simply *!>* and lets me leave the shell without losing the current history list. Note that *!<* in *login.bat* lets me reenter the shell without losing the list. You could also use *!<* (without the *!>*) to read in a list of commonly used commands when the shell boots.

```
setenv PROMPT=[\$s:\$!]
alias a alias
a h history
a book          cd /text/book
a ddj           cd /text/ddj
a here          set here = \$p
a sclass        cd /src/class
a there         cd \$here
a tmac          cd /lib/tmac
a tools         cd /src/tools
a type          cat
a m             "rm err; make -B builtins.mak"
!<
```

Table 1: *A login.bat file*

Shell Invocation Syntax

There are several ways to get into the shell from the command line. The easiest way is to type *sh* (with no arguments), putting you into *interactive* mode. You can't get out of the lowest-level interactive shell with a ^C. Use either *exit* or *logout*.

sh −c string—invokes the shell in nonresident mode. It will execute the command contained in *<string>* as if it had been entered from the command line and then terminate.

sh filename args . . . —executes a batch file. *$0*, if found inside the batch file, will be expanded to the *file name*. The arguments can be fetched with *$1*, *$2*, etc. *%* can be used instead of *$* if you like.

Four other command-line switches are available:

−i—puts the shell into interactive mode even if arguments are listed on the command line. Normally, if command-line arguments are present and −c isn't specified, the shell will try to execute a batch file. *sh −i arg . . . arg* will create an interactive shell, putting the arguments into *$1*, *$2*, etc. *$0* will hold the string *−i*.

−q—causes quotes to be stripped from commands before they're passed to a child process. The quotes will still protect wildcard characters, etc., from expansion.

−v—verbose mode, commands are echoed to *stderr* as they're read by the shell. This is the same as a *set verbose* command.

−x—commands are echoed to *stderr* just before they're executed. All *$* arguments and wildcard characters will have been expanded at this point. This is the same as a *set echo* command.

Redirection

The shell itself doesn't support redirection; however, because command.com is still resident, redirection is available if you need it. There are two ways to use command.com for redirection. A nonresident shell can be invoked from DOS with a line such as:

```
sh -c grep pattern *.c >foo
```

Grep will be executed, and the shell will expand the **.c* to the names of all files in the current directory having a .c extension before *grep* is invoked. *Grep*'s output will be redirected to *foo* in the normal way. Because this command is executed from MS DOS and not from the shell, it won't be added to the history list.

The second method also lets you enter redirected commands into the history list. From inside the shell, type:

```
command /c grep pattern *.c >foo
```

The */c* argument to command.com works like the *−c* argument to *sh*, so it will execute the following string as if it had been typed. The **.c* will again be expanded by the shell before command.com is executed, but the > will be interpreted by command.com (which will put the output into *foo*). Be careful here of command-line truncation. Because command.com doesn't know about the CMDLINE environment, it has no way to get to the extended command line, so it will work on only the first 127 characters.

Support Routines

In order to minimize the size of the shell, I've tried not to build in commands that aren't essential. I've found the following programs to be useful.

cat.c—prints (concatenates) files to *stdout*.

cp.c—copies a file to another file or disk. Copies a group of files to another directory or disk.

echo.c—echoes command line to *stdout*.

grep.c—searches for pattern in file.

ls.c—lists a directory.

mkdir.c—creates a directory.

mv.c—renames a file or moves a group of files to another directory.

printenv.c—prints the current environment.

rm.c—removes a file or group of files.

rmdir.c—removes a directory.

Availability

This column is part of a four-part series describing the entire shell. A reprint of all four parts along with a disk containing the listings is available for $29.95 from *Dr. Dobb's Journal*, 2464 Embarcadero Way, Palo Alto, CA 94303. Please direct inquiries to the The Shell. Prepayment is required. **DDJ**

(Listing begins on page 84)

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 1.**

PL/68K
C Becomes 68000 Assembly Language

by Edward K. Ream

> ***I asked myself, suppose the program produced by the C compiler and the program produced by the assembler were semantically equivalent? Suddenly PL/68K became not just another assembly language but a new way of using C.***

One day not long ago, I became embroiled in an old debate with another programmer named Charlie. . . .

"The programming team I manage is about to start a big project," I said, "and I must decide which language to use."

"Really? Which languages are you considering?"

"C and 68000 assembly language. The product will have strong competition, and great performance is crucial, so it's reasonable to consider assembly language. On the other hand, C is so much easier to use."

"Why don't you program in C and recode in assembly language as needed?" Charlie asked.

"Of course I've considered that. It might work as far as execution speed is concerned, although I'm not sure. C doesn't let you allocate registers globally, and that's a big handicap. Speed is not the only problem, though. The code must be compact, but our C compiler produces code that is 50 percent larger than assembly language. No, there's no doubt about it—eventually the program will have to be written in assembly."

"Do the initial prototyping in C. That's the right way," Charlie persisted. "When the program is finished, recoding in assembly language will be much easier."

"Hmmm. I'm not convinced. Recoding is going to be expensive; we'll end up debugging the whole program twice. There might even be pressure from higher management not to recode and come out with an inferior product."

Charlie just snorted and walked away, muttering something about assembly language being a throwback to the Dark Ages.

Writing in Both C and Assembly

Fortunately, my friend John overheard this conversation. John and I have worked together for 15 years, and we enjoy discussing problems that come up on the job. John laughed, "Charlie is more interested in being right about C than in solving your problem."

"You sound more sympathetic."

"Well, your choice is crucial. Which language you use determines, to a large extent, how your project will turn out."

"Yes. What bothers me most is that I've got to choose now, but I won't know until the project is almost over whether the choice was correct."

"I think I know a way around this dilemma—it's a language I invented called PL/68K."

"John, my only options are C and 68000 assembly language."

"Don't be fooled by the name. PL/68K isn't really an independent language but a way of using C to do assembly-language programming."

"John, you are not making sense!"

"Let me explain. You can think of PL/68K as being either C or assembly language—either/or. But in fact, you can run a program written in PL/68K through both the PL/68K assembler and any standard C compiler. PL/68K is both C and assembly language at the same time."

"Wait a minute. You are going much too fast," I said. "First of all, you can't possibly compile an assembly-language program with a C compiler! Assembly language doesn't look anything like C—the C compiler will spit out a thousand error messages!"

"PL/68K doesn't resemble 'traditional' assembly language. Forget what assembly language usually looks like and ask yourself, 'What are the characteristics of assembly language?' "

"Go on," I replied. "You tell me."

"First, assembly language allows full access to all machine resources—all registers, all locations in memory (including the run-time stack), all I/O ports, all privilege modes, and all machine instructions. Second, there is a one-for-one correspondence between the source code you write and the object code produced by the assembler. You always know what code a particular assembly-language construct generates; assemblers neither rearrange code nor 'optimize' code away nor add anything extraneous. Assemblers are very literal-minded. Thus, assembly language ensures zero time and space overhead."

"You're saying that assembly language gives you complete control over the machine, without a compiler getting in the way."

"Exactly. Now, suppose we say assembly language is any language that (1) allows complete access to all machine resources, (2) provides a clear correspondence between source code and object code, and (3) imposes zero time or space overhead."

Semantic Identity

"Hmmm," I mused. "This definition doesn't say what assembly language looks like. It could even look like C. But I still don't understand. If you run a PL/68K program through an assembler, you will get one program. If you run the same source through a C compiler, you will get a second program. The two programs are not going to do the same things—similar things, maybe, but not the same things. The fact that the source code is the same doesn't matter. To put it another way, given a result desired from a specific PL/68K program, we would still have to choose between assembling the program with the PL/68K assembler or compiling it with a C compiler."

"You've stated the problem very well," John said, "but I have discovered that it's possible to design PL/68K so that the program produced by the PL/68K assembler will work in the same way as the program produced by the C compiler."

"That sounds impossible!"

"I don't think so. Let's turn the problem around. Suppose we design PL/68K according to what might be called 'the principle of semantic equivalence.' This principle states that a program, when assembled by the PL/68K assembler, must work in the same way as when it is compiled by a standard C compiler. Now let's ask, 'What needs to be eliminated from PL/68K to guarantee semantic equivalence?' " (See Figure 1, page 40.)

"Tell me," I said, "how much of C is left after the principle of semantic equivalence takes its toll?"

"Surprisingly, almost all of it. The preprocessor is identical to the C preprocessor. All declarations and structure statements are present. Functions do not return values but otherwise are unchanged, as are Boolean and relational operators and expressions. The biggest restriction is that arithmetic operators and expressions must be severely curtailed in order to make PL/68K expressions mean the same thing as C expressions."

"You keep talking about PL/68K being assembly language," I said. "How is it possible to produce code the quality of assembly language from a language that is a subset of C?"

"I haven't shown you the whole language yet. Two other rules guide the design of PL/68K. These rules, together with the principle of semantic equivalence, determine the form and content of PL/68K. The two rules are 'the code selection rule'—the assembler for PL/68K does no code selection, and all arithmetic operations in PL/68K correspond to unique 68000 machine instructions; and 'the register allocation rule'—the assembler for PL/68K does no register allocation, and all operations in an assignment statement are performed in the location specified by the left side of that assignment statement.

"In short, these rules say that an assembler for PL/68K never has to make any significant decisions. Because the PL/68K assembler knows how to select code and allocate registers, it will never need any of the fancy techniques used by optimizing compilers, but it will be able to produce code that is just as good as assembly language.

"I like to think of the assembler for PL/68K as a simple compiler, consisting of a parser and straightforward code generator and possibly a peephole optimizer. The whole job should take about a year to complete rather than the 10 to 20 programmer years for a typical optimizing compiler. Using compiler technology to write an assembler was the initial idea that started me thinking about PL/68K."

Now that I've presented the general ideas behind PL/68K, I'll drop this dialogue format and these fictional characters and look at the details of the language.

Specifying Registers

Because PL/68K is both assembly language and C, some way must be found to deal with assembly-language constructs such as registers, address modes, and individual machine instructions, while at the same time remaining compatible with C. These assembly-language constructs are represented by reserved words, shown in Table 1, page 29.

As for the registers of the 68000, d0 through d7 stand for the data registers, a0 through a7 for the address registers, pc for the program counter, ssr for the status regis-

PL/68K
(Continued from page 27)

ter, and ccr for the condition code register, which is the lower byte of the ssr.

Standard aliases are also defined. Register a7 can also be called sp, ssp, or usp to denote the stack pointer (or system stack pointer or user stack pointer). The reserved words r0 through r7 are synonyms for d0 through d7, and the names r8 through r15 are synonyms for registers a0 through a7.

All registers on the 68000 are 32 bits long (except the status registers), but not every instruction uses all 32 bits of a register. Besides long (32-bit) operations, byte-length (8-bit) and word-length (16-bit) operations are permitted on data registers, and word-length operations are permitted on address registers. To represent the length of an operation, the name of any data register can be followed by a *b* to denote byte length or a *w* to denote word length. Thus, *d0* stands for the long register d0, *d0w* stands for the word-length register d0, and *d0b* stands for the byte-length register d0. Address registers are treated in a similar manner, except that byte-length operations are not permitted.

Address Modes

The 68000 has 12 different address modes, or means of accessing operands. (See Table 2, page 29.) The address modes are represented in PL/68K by five of C's operators, namely &, *, ++, --, and ->.

Let's look, for example, at the Address Register Indirect with Postincrement address mode. (It's a lot easier to use than to say.) This mode uses the contents of an address register as the address of an operand. After the operation is performed, the address register is incremented by 1, 2, or 4, depending on the size of the operation. In traditional assembly language, that mode applied to address register a0 would be written as *(a0)+*. In PL/68K, that address mode is represented by **a0++*. For example, you would write *d0b=*a0++;* in PL/68K instead of *move.b (a0)+,d0b*. Constructions such as **++a0* are not allowed because of the code selection rule. The 68000 has no addressing mode of the form *+(a0)*, so **++a0* is not part of PL/68K.

The word *primitive* denotes what is called an effective address in machine-language terms. A primitive describes an operand, which may be in a register, on the run-time stack, or in static memory. In PL/68K, the valid forms of primitives are determined by the address modes I've just discussed.

Declarations

While I am talking about operands, I'll say a few words about how those operands are declared. Declarations in PL/68K are just the same as in C, except that functions do not have types. If you think about it for a moment, this means that PL/68K declarations have no parentheses. Declarations can never become unreadable as they can in full C.

In effect, declarations produce DC (define constant) and DS (define storage) pseudo-operations. (See Table 3, page 29.) Although declarations produce no executable code,

they determine what code gets produced by arithmetic operators. For instance, if *a* is an integer, the assignment statement *a* *= *b*, which multiplies *a* by *b*, generates a MULS (signed multiply) machine instruction, but if *a* is an unsigned integer or pointer, the assignment statement generates a MULU (unsigned multiply) instruction.

As another example, the assignment *a* += *b*, which adds *b* to *a*, generates an ADD.B (byte length add) instruction if *a* is a *char*, but it generates an ADD.W (word length add) instruction if *a* is an *int* and generates an ADD.L (word length add) instruction if *a* is a long word or a pointer.

Assembly-Language Instructions and Pseudo-operations

Reserved identifiers also stand for 68000 machine-language instructions and pseudo-operations. A library of pseudofunctions must be linked with a PL/68K program when it is translated with a C compiler. This library, called the ops library, contains declarations and functions that allow C programs to simulate the effect of 68000 machine instructions and pseudo-operations. (See Table 4, page 32.)

The pseudofunction *btst( )*, for example, simulates the BTST (bit test) machine instruction. In PL/68K, you would write *btst(1,d0b);* in those places where you would write *btst.b #1,d0* in traditional 68000 assembly language.

Other pseudofunctions allow PL/68K programs to refer to assembly-language pseudo-operations. The PL/68K assembler translates the *org( )*, *even( )*, *bss( )*, *text( )*, and *data( )* pseudofunctions to the ORG, EVEN, BSS, TEXT, and DATA pseudo-operations. Similarly, the PL/68K assembler translates the *dcb( )*, *dcw( )*, *dcl( )*, *dsb( )*, *dsw( )*, and *dsl( )* pseudofunctions to the DC.B, DC.W, DC.L, DS.B, DS.W, and DS.L pseudo-operations. None of these pseudofunctions has any effect when a C compiler translates a PL/68K program. In other words, the corresponding pseudofunctions in the ops library do nothing.

You may be wondering why I keep calling these routines pseudofunctions. After all, they are perfectly good functions when compiling a PL/68K program with a C compiler. When you turn the program through the PL/68K assembler, though, it translates pseudofunctions directly into 68000 machine instructions or pseudo-operations.

Expressions and Assignment Statements

I've covered the components of low-level assembly language—registers, address modes and effective addresses, machine instructions, and pseudo-ops. Now let's see how you put these components together to make expressions and assignment statements.

Expressions are quite restricted; they are just primitives or parenthesized constant expressions. Assignments are restricted to the forms *primitive aop expression* or *primitive aop (assignment)*, where *aop* stands for one of the assignment operators of the C language, namely, =, +=, −=, *=, /=, &=, |=, ^=, >>=, and <<=. The regular arithmetic operators in C, namely, +, −, *, /, &, |, ^, >>, and << are allowed only in constant expressions, which must be parenthesized.

You are probably wondering why all these restrictions exist. There's a short answer and a long answer. The

Data registers
d0, d0b, d0w, . . . , d7, d7b, d7w
r0, r0b, r0w, . . . , r7, r7b, r7w

Address registers
a0, a0w, . . . , a7, a7w
r8, r8w, . . . , r15, r15w
sp, spw, usp, uspw, ssp, sspw

Status registers
ssr, ccr

Program counter
pc

Table 1: *Reserved words corresponding to 68000 registers*

PL/68K	Traditional assembly language
123	#123
abc	abc
&abc	#abc
*(&abc+1)	abc+#1
abc.25	abc+#25
*(0x80)	$80
a0	a0.l
a0w	a0.w
d0	d0.l
d0w	d0.w
d0b	d0.b
*a0	(a0)
*a0++	(a0)+
*−−a0	−(a0)
a0 → 5	#5(a0)
a0 → d0	#0(a0, d0.l)
a0 → (d0w)	#0(a0, d0.w)
a0 → (d0+5)	#5(a0, d0.l)
a0 → (d0w+5)	#5(a0, d0.w)
pc → 5	#5(pc)
pc → d0	#0(pc, d0.l)
pc → (d0w)	#0(pc, d0.w)
pc → (d0+5)	#5(pc, d0.l)
pc → (d0w+5)	#0(pc, d0.w)

Table 2: *Representing the address modes of the 68000*

Declaration	Code Generated
char abc;	abc: ds.b 1;
char c1 = 'c';	c1: dc.b 'c';
char * cp;	cp: ds.l 1;
long xyz;	xyz: ds.l 1;
char a [ ] = "abc";	a: dc.b 'abc',0;
int a2 [25];	a2: ds.w 25;
struct s1 {	
long sl;	
char *s2;	
int s3;	
};	no code generated.
struct s1 s2 [25];	s2: ds.b 250;
union u1 {	
int ui;	
char uc;	
long ul;	
};	u1: ds.b 4;

Table 3: *Code generated for declarations*

(Continued from page 29)

short answer is that it's the only way to reconcile all of the design rules behind PL/68K with all of C's rules concerning operator precedence and arithmetic conversions. Take my word for it: You cannot have full C expressions in PL/68K and still retain the principle of semantic equivalence. For the long answer to this question, see the Appendix section.

There are additional restrictions concerning assignments to pointers. Suppose, for example, that *a0* has been declared to be a pointer to some object *X* whose size is larger than 1. In this case, the only assignments allowed to *a0* are assignments of the form

```
a0 += constant;
```

(or constant expression) or

```
a0 -= constant;
```

To conform to C's scaling rules, these constructions are equivalent to

```
ADDA CONSTANT * SIZEOF(X), A0
SUBA CONSTANT * SIZEOF(X), A0
```

If you do not want this scaling to take place, you can use the *adda( )* or *suba( )* pseudofunctions. In general, no scaling, optimization, or other kind of tampering is ever done to the arguments of pseudofunctions. This rule has no exceptions.

Two special forms of assignment statements are permitted: *location++;* and *location--;*, where *location* stands for an address register, data register, or the name of a memory variable.

A few words about why PL/68K does not have function calls and array references—let's consider function calls first. Functions don't return values in a designated location because no single spot is right in all cases. Returning the results of a function in a register would conflict with the register allocation rule. Thus, the programmer must specify how functions will return values. Note, however, that you can pass arguments to a function in the normal way, as you shall see shortly.

The array operator *[ ]* conflicts with both the register allocation and the code selection rules. There is no suitable spot in which to evaluate subscript expressions, and picking a spot at random would violate the register allocation rule. (Actually, no single location would suffice because general C expressions can require arbitrarily many temporary locations to evaluate.) In addition, the "natural way" to access arrays on the 68000 is with pointers, which is how an assembly-language programmer would probably do it. Thus, retaining the array operator would violate the spirit behind the code selection rule.

Assignment statements are simple, but that shouldn't hurt much—assignment statements usually have only one or two operators anyway.

As the code selection rule requires, PL/68K defines what code arithmetic expressions will generate. (See Table 5, page 32.) The += operator generates the ADD instruction, the -= operator generates the SUB instruction, and so on. Many machine instructions on the 68000 have several variants—for example, ADDA adds address registers, ADDI adds literal data, and plain ADD adds to data registers and memory locations. The assembler has to do some code selection, but the choice is easy; even traditional assemblers do that much.

Boolean Expressions

Boolean expressions in PL/68K are similar to Boolean expressions in C. All the Boolean operators *!*, *||*, and *&&* and all the relational operators ==, !=, <, <=, >, and >= are allowed. Of course, arithmetic expressions are restricted in Boolean expressions just as they are in assignment statements.

Boolean expressions in PL/68K are not general expressions, as they are in C. Boolean expressions can only appear in the appropriate part of *if*, *do*, *while*, and *for* statements. The code fragment *a == b;* is not valid in PL/68K outside a structure statement. This restriction eliminates a whole class of hard-to-find errors (the programmer almost certainly meant to say *a = b;*).

Because Boolean expressions appear in limited contexts, much less work is required to evaluate them. (See Tables 6 and 7, page 32.) Boolean expressions of the form *primitive* or *primitive relop 0* or *0 relop primitive* (where *relop* denotes one of the relational operators) generate the TST (test operand) instruction followed by some form of the *Bxx* (conditional branch) instruction. The *Bxx* instruction chosen depends on the *relop* and the declared type of the operand being tested.

Expressions of the form *primitive relop nonzero constant* or *nonzero constant relop primitive* generate the CMP (compare) instruction followed by a *Bxx* instruction.

The *NOT* operator generates no code at all but instead simply reverses the "polarity" of one or more *Bxx* instructions. For example, the statement

```
if (a == 0)
```

generates

```
TST A
BNE
```

while the statement

```
if (!(a == 0))
```

generates

```
TST A
BEQ
```

Similarly, the || and && operators generate no extra code. Incidentally, you can use parentheses in Boolean expressions to affect the order of binding of Boolean operators. For instance,

```
if (a && !(b < 5 || b > 20))
```

A FULL C COMPILER FOR $49.95

Ecosoft's Eco-C88 C Compiler

An unbeatable value! For $49.95 you get:

- A C compiler with all data types and operators (except bit fields)
- Fast executing code. Some common benchmark results are:

	Eco-C88	L (1)	C86 (1)	MS (1)	MW (1)
sieve	12	11	13	11	12
fib	43	58	46	109	—
deref	14	13	—	10	11
matrix	22	29	27	28	29

1. Computer Language, Feb., 1985. Reproduced with permission.

- 8087 support using a single library. The 8087 is sensed at runtime and used if present.
- A standard library with over 200 functions (many of which are System V compatible for greater portability).
- Error messages in English - no cryptic numbers to look up.
- A cc and "mini-make" (in source) that makes compiling a snap.
- ASM or OBJ output (for use with MSDOS linker).
- Expanded user's manual.
- Works with all IBM PC's and clones using MSDOS 2.1 or later.

ORDERS ONLY 1-800-952-0472

The ECOSOFT family of C products ORDER FORM

☐ C Compiler $49.95 ________
☐ Programming Guide $20.00 ________
☐ Self-Study Guide $17.00 ________
☐ Programmer's Library $20.00 ________
☐ Program Editor $29.95 ________
☐ C Library Source $10.00 ________
☐ ISAM (.OBJ) $15.00 ________
Total* (Ind. res. add 5% tax) ________
*Please add $4.00 for shipping.
Payment: VISA MC AE Check
Credit card expir. date ________
Card # ________
Name ________
Address ________
City, state ________
Zip ________ Phone ________

Ecosoft, Inc.
6413 N. College Ave.
Indianapolis, IN 46220
(317) 255-6476 · 8:30-4:30

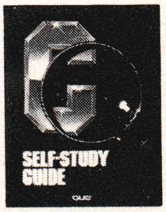
ECOSOFT INC.

Other Eco-C Products

CED Program Editor. $29.95
A screen-oriented program text editor similar to the Turbo Pascal editor. You can create, compile and link the source file with CED. If there is an error, CED automatically reloads the source file and places the cursor on the offending section of code. CED supports editing multiple files (with windows), macros, more than 50 editing commands and is configurable to your particular needs and preferences. An outstanding value.

C Programming Guide, 2nd Ed. (Purdum, Que Corp.) $19.95
This best seller walks you through the C language in an easy-to-read manner. All aspects of the language are covered, including many of the new ANSI Standards suggestions. Many of the error messages issued by the Eco-C88 compiler reference page numbers in this text making an ideal learning environment.

C Self-Study Guide (Purdum, Que Corp.) $16.95
This new book is designed for the person that is learning C on their own. The book is filled with questions and answers that most beginning C programmers have. It also includes many sample programs that illustrate tips, traps and techniques that may take years to discover otherwise. A perfect compliment to the Guide book.



C Programmer's Library (Purdum, Leslie, Stegemoller, Que Corp.) $19.95
This best seller is an intermediate level text designed to show you how to write functions for your C library. The book contains many useful library additions, including an ISAM file handler, plus sections on advanced C topics.

TRADEMARKS: ECO-C88, ECOSOFT; TURBO PASCAL, BORLAND INT'L.

Pseudofunctions corresponding to 68000 machine instructions

abcd, add, adda, addi, addq, addx, and, andi, asl, asr
bcc, bcs, bchg, beq, bge, bgt, bhi, ble, bls, blt, bmi, bne, bpl, bvc, bvs
bchg, bclr, bra, bset, bsr, btst
chk, clr, cmp, cmpa, cmpi, cmpm,
dbcc, dbcs, dbeq, dbf, dbge, dbgt, dbhi, dble, dbls, dblt, dbmi, dbne, dbpl, dbt, dbvc, dbvs, divs, divu
eor, eori, exg, ext, jmp, jsr, lea, link, lsl, lsr
move, movea, movem, movep, moveq, muls, mulu
nbcd, neg, negx, nop, not, or, ori, pea
reset, rol, ror, roxl, roxr, rte, rtr, rts
scc, scs, seq, sf, sge, sgt, shi, sle, sls, slt, smi, sne, spi, st, svc, svs
sbcd, stop, sub, suba, subi, subq, subx, swap
tas, trap, trapv, tst, unlk

Pseudofunctions corresponding to assembly-language pseudo operations

dcb, dcw, dcl, dsb, dsw, dsl
org, data, text, bss, even

Table 4: *Pseudofunctions*

Operator	Generated code		
a += b	add	b, a	(or adda or addi or addq)
a −= b	sub	b, a	(or suba or subi or subq)
a *= b	muls	b, a	(or mulu)
a /= b	divs	b, a	(or divu)
a %= b	divs	b, a	(or divu)
	swap	a	
a >>= b	asr	b, a	(or ror)
a <<= b	asl	b, a	(or rol)
a &= b	and	b, a	(or andi)
a |= b	or	b, a	(or ori)
a ^= b	eor	b, a	(or eori)
a ++	addq	#1, a	(or addi)
a −−	subq	#1, a	(or subi)

Table 5: *Code generated by arithmetic operators*

Boolean	Generated code		
if (Z)	bnz	*false*	
if (a)	tst	a	(or cmpa)
	bz	*false*	
if (a < b)	cmp	b,a	
	bge	*false*	
if (!a)	tst	a	
	bnz	*false*	
if (a && b)	tst	a	
	beq	*false*	
	tst	b	
	beq	*false*	
if (!(a && b))	tst	a	
	beq	*true*	
	tst	b	
	bne	*false*	
	true:		

Table 6: *Code generated by Boolean expressions*

Signed Comparisons			Unsigned Comparisons	
		if (c1 == c2)		
cmp	c2,c1		cmp	c2,c1
bne	*false*		bne	*false*
		if (c1 != c2)		
cmp	c2,c1		cmp	c2,c1
beq	*false*		beq	*false*
		if (c1 < c2)		
cmp	c2,c1		cmp	c2,c1
bge	*false*		bcc	*false*
		if (c1 <= c2)		
cmp	c2,c1		cmp	c2,c1
bgt	*false*		bhi	*false*
		if (c1 > c2)		
cmp	c2,c1		cmp	c2,c1
ble	*false*		bls	*false*
		if (c1 >= c2)		
cmp	c2,c1		cmp	c2,c1
blt	*false*		blo	*false*

Table 7: *Code generated by Boolean comparisons*

Macro Name			Pseudo-function	Meaning
Z	or	EQ	cc_z	zero
NZ	or	NE	cc_nz	not zero
C	or	CS	cc_c	carry
NC	or	CC	cc_nc	no carry
V	or	VS	cc_v	overflow
NV	or	VC	cc_nv	no overflow
GT			cc_gt	greater than
GE			cc_ge	greater or equal
LS			cc_ls	less than
LE			cc_le	less than or equal
HI			cc_hi	high
LO			cc_lo	low
MI			cc_mi	minus
PL			cc_pl	plus

Table 8: *Condition code constants*

Syntax

(1) if (*boolean*) { *statement list* }
(2) if (*boolean*) { *statement list 1* } else { *statement list 2* }

Code generated for (1)

$ *Evaluate boolean. If false, jump to label 1* $
$ *statement list* $
label1:

Code generated for (2)

$ *Evaluate boolean. If false, jump to label 1* $
$ *statement list 1* $
bra label2;
label1:
$ *statement list 2* $
label2:

Table 9: *The if statement*

PL/68K
(*Continued from page 30*)

is valid and generates

```
TST A
BEQ
CMPI 5, B
BLT
CMPI 20, B
BGT
```

You can specify condition code values directly. (See Table 8, page 32.) For instance, the statement *if (Z)* tests the current value of the zero bit in the condition code register and generates the BNZ (branch not zero) instruction. *Z* is a macro in C, defined in the ops library, which expands to a call to the pseudofunction *cc_z( )*.

Structure Statements

I said earlier that PL/68K has all C's structure statements—*if, do, while, for,* and *switch*. They look exactly like C code, but curly braces are required surrounding statement lists in structure statements. In other words, structure statements have the form

```
if(. . .) {statement list}
if(. . .) {statement list} else {statement list}
while(. . .) {statement list}
do {statement list} while (. . .);
for (. . .) {statement list}
switch(. . .) {statement list}
```

In my opinion, allowing curly braces to be optional is a big flaw in C. In this example:

```
if (abc < 5)
    xyz = 5;
    if (abc < 6)
        xyz = 6;
```

the indentation is misleading and will probably cause a bug. This kind of error can be extremely difficult to find.

Let's see what code PL/68K's structure statements produce. In the accompanying tables, the dollar sign denotes code that corresponds to some language construct. For example, *$ statement list $* stands for whatever code is generated for the statement list. The statement list could be arbitrarily complicated—for instance, it could contain nested structure statements. The notation

$ Evaluate boolean. If false, jump to label1 $

indicates that code is generated for the Boolean expression such that a jump to *label1* is taken if the Boolean expression is false. Otherwise, control falls through to the following code. Labels are indicated in the usual way, by identifiers followed by colons. All generated labels are, of course, unique, even though they may have identical names in the tables.

Table 9, page 32, shows the *if* statement. When an *if* statement contains no *else* clause, the Boolean expression

Syntax

```
(1) while ( boolean ) { statement list }
(2) while (1) { statement list }
```

Code generated for (1)

```
bra continue_label;
label1:
$ statement list $
continue_label:
$ Evaluate boolean. If true, jump to label 1 $
break_label:
```

Code generated for (2)

```
continue_label:
$ statement list $
bra continue_label;
break_label:
```

***Table 10:** The while statement*

Syntax

```
(1) do { statement list } while ( boolean );
(2) do { statement list } while(1);
```

Code generated for (1)

```
label1:
$ statement list $
continue_label:
$ Evaluate boolean. If true, jump to label1 $
break_label:
```

Code generated for (2)

```
continue_label:
$ statement list $
bra continue_label;
break_label:
```

***Table 11:** The do statement*

Syntax

```
(1) for ( assignment list 1; boolean; assignment list 2)
      { statement list }
(2) for ( assignment list 1 ; ; assignment list 2 ) { statement list }
(3) for ( assignment list 1 ; 1; assignment list 2 ) { statement list }
```

Code generated for (1)

```
$ assignment list 1 $
bra label0;
label1:
$ statement list $
continue_label:
$ assignment list 2 $
label0:
$ Evaluate boolean. If true, jump to label1 $
break_label:
```

Code generated for (2) or (3)

```
$ assignment list 1 $
label1:
$ statement list $
continue_label:
$ assignment list 2 $
bra label1;
break_label:
```

***Table 12:** The for statement*

PL/68K

(Continued from page 33)

is evaluated, and control either falls through to the *then* clause or a jump is made to the end of the statement.

Similar code is generated when the *if* statement contains an *else* clause. The Boolean expression is evaluated, and control either falls through to the *then* clause or a jump is made to the *else* clause. A BRA (branch always) instruction following the *then* clause skips around the *else* clause.

The code generated for the *while* statement, shown in Table 10, page 34, might be a little controversial. The first instruction is a branch to the end of the loop so that the loop test occurs at the bottom. This produces the fastest code unless the *while* loop is executed less than once on average. In the rare cases in which this jump is unwanted, the programmer must simulate the loop in some way.

Notice the labels called *continue_label* and *break_label*. These are used as target labels for the *break* and *continue* statements. In other words, within a *while*, *do*, or *for* statement, the effect of a *continue* instruction is to generate a branch to the *continue_label* defined for that statement. Similarly, a *break* statement generates a jump to the appropriate *break_label*. As in C, you can also use the *break* statement inside a *switch* statement.

The *while* statement generates different code if the Boolean expression is a nonzero constant. This is a common idiom in C, and the definition of PL/68K ensures that there is no time penalty for using it.

The code for the *do* statement, shown in Table 11, page 34, is similar to the code produced by the *while* statement. The code for the *for* statement (see Table 12, page 34) is more interesting. If the loop test in a *for* statement is nontrivial, the code for it appears at the bottom of the loop. Note also that the syntax of the *for* statement is more restricted than in C.

The *switch* statement, shown in Table 13, page 37, generates a jump table—i.e., a table of addresses. Code is generated that jumps through that table to the proper *case* statement, based on the contents of a register. Note that the *switch* statement destroys this register.

It is sometimes better to generate a sequence of tests rather than a table jump, but the *case* statement always generates a table jump. Remember, each language construct in PL/68K stands for a particular sequence of code—if you want a sequence of tests, use a sequence of *if* statements; if you want a table jump, use a *switch* statement.

Many compilers generate jumps to jumps when they translate structure statements, but the definition of PL/68K requires that all jumps to jumps (and jumps to return statements) be eliminated. The assembler can do this in several ways. For instance, if the assembler creates a parse tree for an entire function before any code is generated, it's easy for the code generator to look at the target of any jump to see if it is another jump or a return instruction. Alternatively, the assembler can use a standard peephole optimizer.

Function Calls

Functions in PL/68K can have formal parameters and lo

PL/68K
(Continued from page 34)

cal arguments, just as in C. The code shown in Table 14, page 37, is generated by function calls. Code is generated to push all arguments on the stack, a JSR (jump to subroutine) instruction is generated, and, if necessary, an ADD instruction is generated to pop arguments off the stack. One long word is always reserved on the stack for the first argument, which shortens the calling sequence when there are less than two arguments.

The ADD instruction can be eliminated by having the called program, instead of the calling program, pop the arguments off the stack, but the sequence shown in Table 14 is the fastest. If you eliminated the ADD instruction and pushed the arguments in the same way (that is, above the return address), the called program would need to do much more work to pop off its arguments. You could also push the actual arguments below the return address, but that way actually increases the length of the calling sequence. In order to push arguments below the return address, you would have to use an instruction such as *move arg,n(sp)*, which is 2 bytes longer than *move arg,−(sp)*.

Unlike standard C, PL/68K does specify the order in which arguments are pushed, namely in reverse order. Thus, a function that takes a variable number of arguments—*printf( )* for instance—will find its first argument on the top of the stack.

Of course, it is often best to pass arguments in registers, but PL/68K doesn't need a separate mechanism to do this. Suppose you have a function called *g( )* whose two arguments are passed on the stack. To change *g( )* so that it will take its arguments in registers, you just define the following macro

```
#define g(a,b)d0=a; d1=b; g1()
```

and change *g*'s name to *g1*. Notice that a statement such as

```
if (. . .) g(x,y);
```

cannot cause problems in PL/68K because you must write

```
if (. . .) {g(x,y);}
```

instead. (If braces were omitted, after macro expansion, the code would be

```
if (. . .) d0=a;d1=b;g1( );
```

and only the assignment *d0=a* would be part of the *if* statement.)

This macro might generate redundant code. Suppose it were called with *d0* as the first argument, for instance. The macro would expand to *d0=d0* and generate the instruction *move d0,d0*. To handle that problem, the PL/68K assembler eliminates redundant moves. If you must generate such a redundant move for some reason, use a pseudofunction. Pseudofunctions are never second-guessed by the assembler.

Syntax

```
switch ( reg ) {
case constant 1 :    statement list 1 ;
case constant 2 :    statement list 2 ;
. . .
case constant n :    statement list n ;
default:             default statement list ;
}
```

Code generated

```
if (reg < min || reg > max) {
        goto default_label;
}
else {
    $ goto the routine whose address is at table [reg] $
}

table:
dc.l  label i_min;
dc.l  label i_min+1;
. . .
dc.l  label i_max-min+1;

label1:
$ statement list 1 $
label 2:
$ statement list 2 $
. . .
label n:
$ statement list n $
default_label:
$ default statement list $
break_label:
```

***Table 13:** The switch statement*

No arguments

```
    jsr      function
```

One argument

```
    move     arg, (sp)
    jsr      function
```

Two or more arguments

```
    move     arg_n, (sp)
    move     arg_n-1, -(sp)
    . . .
    move     arg_2, -(sp)
    move     arg_1, -(sp)
    jsr      function
    add      # size of all arguments except arg_n, sp
```

***Table 14:** Code generated for function calls*

Entry

```
    movem.l  all_local_registers , -(sp)
    link     a_n, # - size of local auto variables - 4
```

Exit

```
    unlk     a_n
    movem.l  (sp)+, all_local_registers
    rts
```

***Table 15:** Code generated for entry/exit of A_n-based functions*

(Continued from page 36)

Accessing Variables Within Functions

Now that I've covered the passing of arguments to a function, I'll explain how the function gets hold of those arguments. This is a complicated subject, so before getting involved in some messy details, let's handle the easy cases.

First, PL/68K keeps its hands off all registers declared outside any function. These global registers can be accessed from within functions, but PL/68K never generates code to save or restore them. Consequently, you can prevent PL/68K from interfering with any register simply by declaring that register outside a function. By the way, the register keyword is not valid outside functions, but that does not prevent you from declaring register variables anywhere you wish, either in C or in PL/68K. For instance, you can declare *a0* to be global simply by saying *char * a0;* outside any function.

Second, except for these global registers, all registers used in a function are saved on entry and restored on exit from the function. The assembler uses the MOVEM.L (move multiple register, long) instruction for this purpose. Accessing register variables declared in a function is, of course, easy.

Third, if a local variable is declared to be static, memory is allocated to that variable in static memory, not on the stack. No extra code is needed to access that variable, either on function entry or exit.

Static internal variables are not very useful; because the values of static internals are retained between invocations of a function, static internals are destroyed by recursive function calls. Also, on many machines (including the 68000) accessing a variable in static memory is more expensive than accessing a "local auto" variable—that is, a local stack variable. At any rate, generating code for static internal variables is straightforward and is not affected by the following complications.

Functions need to access two kinds of nonregister variables: formal parameters and local auto variables. Both types are allocated on the stack. You have three choices for how PL/68K generates code for these stack variables. You make your choice using one of three pseudofunctions: *base(A_n)*, *base(sp)*, or *nobase( )*. If none of these appears in a function, the default is *base(sp)*.

First, you can have PL/68K access stack variables via an address register that is different from the stack pointer, as shown in Table 15, page 37. If the *base(A_n)* pseudofunction appears anywhere in the function (where A_n is any address register except the stack pointer, *a7*), PL/68K generates a LINK (link and allocate) instruction on function entry and an UNLK instruction on function exit. All stack variables are accessed via the address register named in the *base(A_n)* pseudofunction.

Second, you can have PL/68K access stack variables via the stack pointer, as shown in Table 16, page 38. The assembler generates this kind of code if the *base(a7)* or *base(sp)* pseudofunction appears anywhere in the function. The code generated for *sp*-based functions is more complicated, but more efficient, than that for A_n-based functions. If an *sp*-based function contains no function calls, the stack pointer is not incremented on function entry and local auto variables are accessed using positive offsets from the stack pointer. If the function does contain other function calls, however, space is reserved for local auto variables by incrementing the stack pointer on function entry, and local auto variables are accessed using negative offsets from the stack pointer.

Third, you can access stack variables by hand. If the *no_base( )* pseudofunction occurs anywhere in the function, no code is generated on function entry or exit, declarations of stack variables are ignored, and no explicit references to stack variables are permitted. The *no_base( )* pseudofunction is useful when you must play some kind of game with the stack.

Sp-based functions are somewhat dangerous. If the stack pointer is changed using a pseudofunction, the offsets used to access stack variables are going to get out of sync. To handle this problem, several pseudofunctions, shown in Table 17, page 38, allow you to indicate that the offsets should be changed.

The *push(arg)* and *pop(arg)* pseudofunctions are equivalent to the *move(arg,*−−sp)* and *move(*sp++,arg)* pseudofunctions, but in addition, they tell the PL/68K assembler to adjust the offsets used to access stack variables in *sp*-based functions. The *addsp(n)* and *subsp(n)* pseudofunctions are equivalent to the *addi(n,sp)* and *subi(n,sp)* pseudofunctions, but again they tell the assembler to adjust offsets. Finally, the *adjsp(n)* pseudofunction generates no code but tells the assembler to adjust the offsets.

If one or more functions called within this function

Entry

```
movem.l   all_local_registers , -(sp)
suba      # size of local auto variables + 4, sp
```

Exit

```
adda      # size of local auto variables + 4, sp
movem.l   (sp)+,all_local_registers
rts
```

If no function called within this function

Entry

```
movem.l   all_local_registers , -(sp)
```

Exit

```
movem.l   (sp)+, all_local_registers
rts
```

Table 16: *Code generated for entry/exit of sp-based functions*

Pseudofunction	Meaning	
base(a_n)	use a_n basing for stack variables	
base(sp)	use sp basing for stack variables	
nobase( )	access all stack variables "by hand"	
push(arg)	move arg, −(sp)	and adjust stack offsets
pop(arg)	move (sp)+, arg	and adjust stack offsets
addsp(n)	adda #n, sp	and adjust stack offsets
subsp(n)	suba #n, sp	and adjust stack offsets
adjsp(n)	adjust stack offsets	

Table 17: *Pseudofunctions for stack operations*

(Continued from page 38)

These pseudofunctions have no effect on A_n-based functions.

Summary

To summarize the strengths and weaknesses of PL/68K, the syntactical restrictions that make PL/68K a strict subset of C are listed as follows:

- All operand/operator combinations must correspond to an instruction in the 68000 instruction set.
- Functions neither have types nor return values.
- The array operator *[ ]* is eliminated, but arrays may be declared.
- Assignment statements are severely restricted.
- Statement lists must be surrounded by curly brackets.
- There are no floating points, or double constants, or operators.

There are no additions or changes to C syntax. PL/68K and assembly language are semantically identical; the advantages of PL/68K over assembly language are all syntactical:

- C syntax makes programs easier to read.
- C syntax eliminates many common coding errors.
- Far fewer visible labels are required.

PL/68K is not the successor to C nor is it superior to C for most programming projects. The advantages of PL/68K over C apply only in limited circumstances, but when performance is paramount, PL/68K stands out:

- All machine resources and instructions are available.
- Global allocation of registers is possible.
- Total control over generated code is possible.
- Generated code is smaller and faster.

Figure 2, page 42, shows the relationship between PL/68K and its two parent languages. PL/68K is only a subset of C; not all of C is included. On the other hand, C must be augmented by the ops library in order to simulate all the machine resources of assembly language.

Listing One, page 101, shows an example of a PL/68K function. This function finds a name in a symbol table and returns information about that name in registers. Listing Two, page 101, shows the code generated by the PL/68K assembler for that function.

Conclusion

PL/68K started life as a C-like assembly language for the 68000 chip. High-level assemblers are not new—a paper by Niklaus Wirth (*Journal ACM* 15, January 1, 1968) described a high-level assembly language for IBM 360 machines.

My early thinking about PL/68K was influenced by Wirth's paper and by the register allocation and code selection principles. At that stage, I was interested in using compiler technology to create better assemblers. I saw no particular reason to make PL/68K a subset of C.

I felt dissatisfied with the initial version of PL/68K; it was doomed to be "just another computer language." Not wanting to add another minor dialect to the Babel of computer languages, I decided to make PL/68K's syntax identical to C's. This was a lucky decision.

From the first, PL/68K had to be able to name all machine resources, including machine instructions and assembly-language pseudo-ops. Early versions of the language allowed "raw" lines of assembly code to be interspersed with C-like lines of code. Although this approach had some merit, my decision to make PL/68K's syntax compatible with C convinced me to use functional notation to represent assembly-language features. This change was also fortunate.

I began to speculate about what would happen if a PL/68K program were compiled with a C compiler. I asked myself, suppose the program produced by the C compiler and the program produced by the PL/68K assembler were semantically equivalent? Suddenly, PL/68K became not just another assembly language but a new way of using C.

The principle of semantic equivalence guided a rede-

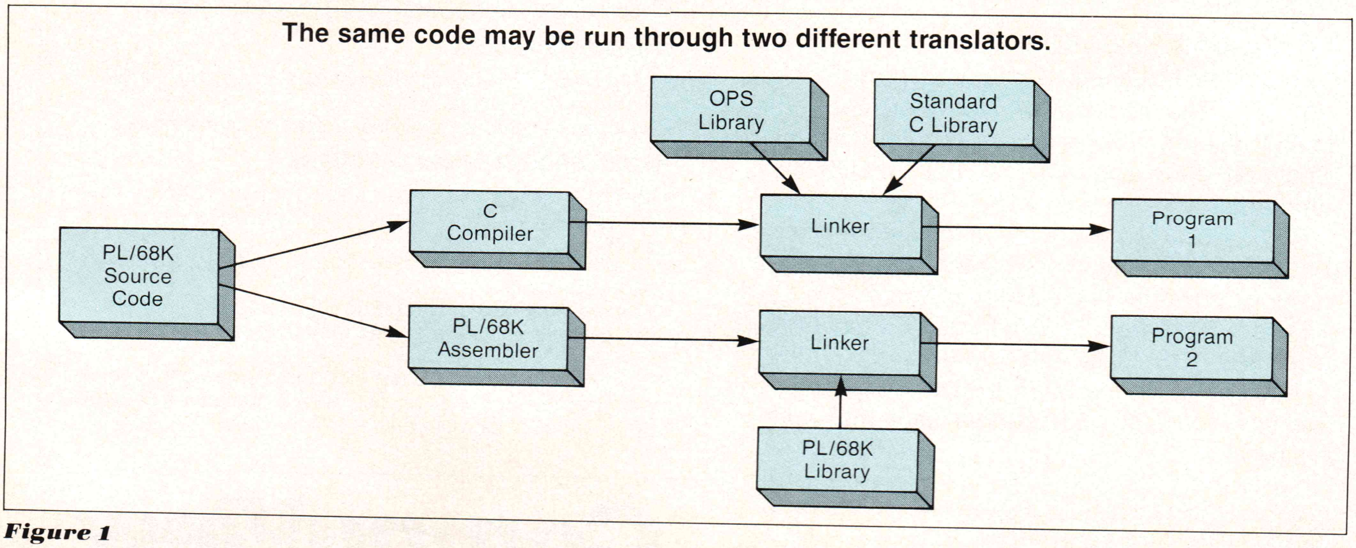
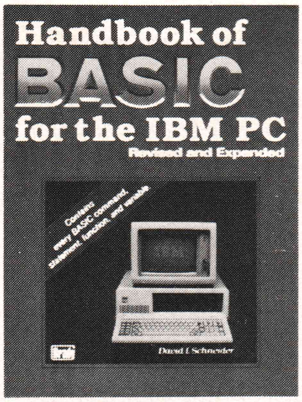


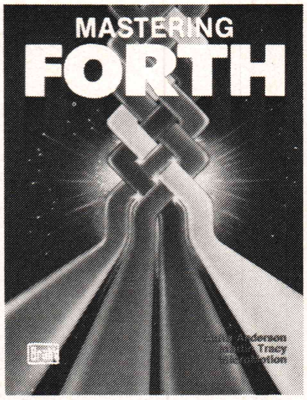
Figure 1

BRADY Speaks Your Language

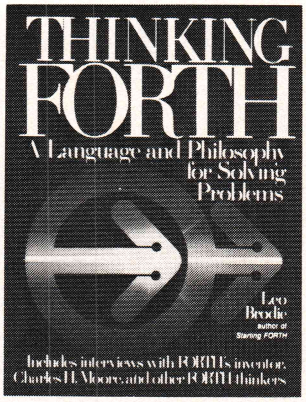
We can help you buff up your BASIC...Put a finish on your FORTH... Conquer C or COBOL...Learn how to LISP...Succeed with Assembler... And more! Just call toll-free or use the coupon to order today!



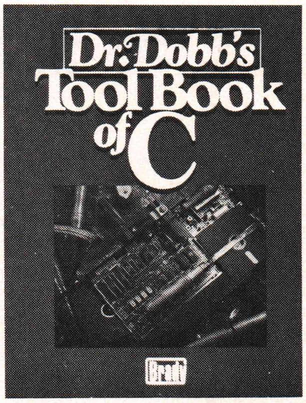
1. *PC* magazine calls it: "A truly remarkable book...A treasure trove of useful programming information for the IBM PC." $22.95



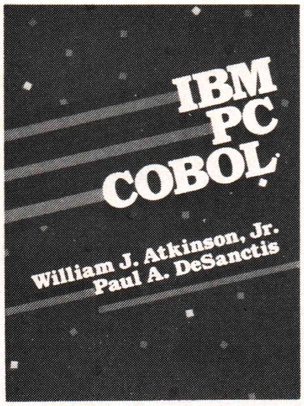
2. A step-by-step tutorial for the high-level, stack-oriented FORTH-83 standard. $17.95



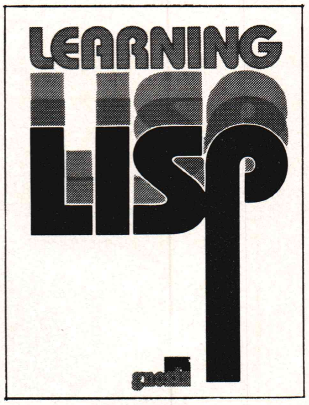
3. The FORTH authority, Leo Brodie, illustrates the elegant logic behind FORTH and shows how to apply specific problem-solving tools to software. $15.95



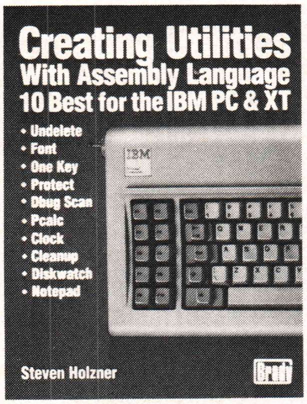
4. The best C articles from the highly respected *Dr. Dobb's Journal* dealing exclusively with C language and programming techniques. $24.95



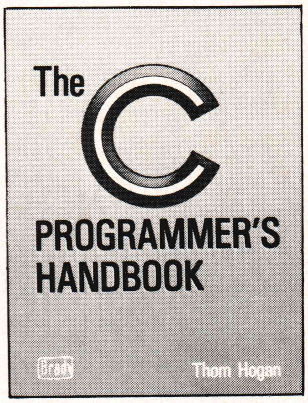
5. The only book of its kind to focus on the differences between micro and mainframe COBOL—and that includes invaluable, fully coded COBOL programs for the IBM PC. $19.95



6. A new guide to LISP (List Processor), the much-talked-about language of artificial intelligence. $14.95



7. For the more advanced user familiar with Assembly language, here's an opportunity to unleash the power of 10 DOS-enhancing programs. $19.95



8. The definitive desktop reference for anyone using the C language—a virtual clearinghouse for C language information. $19.95

Circle **no. 219** on reader service card.

Now at your book or computer store.
Or order toll-free today: **800-624-0023** In New Jersey: 800-624-0024

Brady COMMUNICATIONS COMPANY, INC.
c/o Prentice Hall,
P.O. Box 512, W. Nyack, NY 10994

Circle the numbers of the titles you want below. (Payment must be enclosed; or, use your charge card.) Add $1.50 for postage and handling.
Enclosed is check for $______ or charge to
☐ MasterCard ☐ VISA.

Acc't # ______________ Exp. date ______
Signature ______________
Name ______________
Address ______________
City ______________ State ______ Zip ______
(New Jersey residents, please add applicable sales tax.)
Dept. 3

G-LDDJ-AE(1)

1 (0-89303-510-6) 2 (0-89303-660-9) 3 (0-13-917568-7) 4 (0-89303-599-8)
5 (0-8359-3051-3) 6 (0-13-527813-9) 7 (0-89303-584-X) 8 (0-89303-365-0)

PL/68K

(Continued from page 40)

sign of the language; any construct that violated this principle had to go. In addition, I invented the ops library so that C programs could simulate 68000 machine instructions. Along with the ops library, the concept of pseudofunctions was born, and the language was complete.

PL/68K allows me to sidestep a question that has been haunting me ever since I started programming—whether to program in assembly language and accept the resulting inconveniences or to program in a high-level language and accept a final product that is larger and slower than it could be. PL/68K solves that dilemma.

You can design and write a program in C, keeping in mind the possibility that you will convert it to PL/68K eventually. You can write difficult parts of the program, or parts of the program that will not appear in the final version, using all of C's features. After you have debugged the C program, you can produce a PL/68K version of the program, if desired, by rewriting or excluding those parts of the program that use full C. You then test the PL/68K version and improve its performance as much as you require.

PL/68K hits precisely the right level of abstraction for systems programming. All the features of C allow you to design and code programs easily, but when you need to do low-level work, nothing stands in your way.

A final thought—you could transfer most of the syntax and all the design rules of PL/68K to a similar language for other machines. In this limited sense, PL/68K is a machine-independent assembly language—PL/68K programs are much more portable than programs written in traditional assembly language.

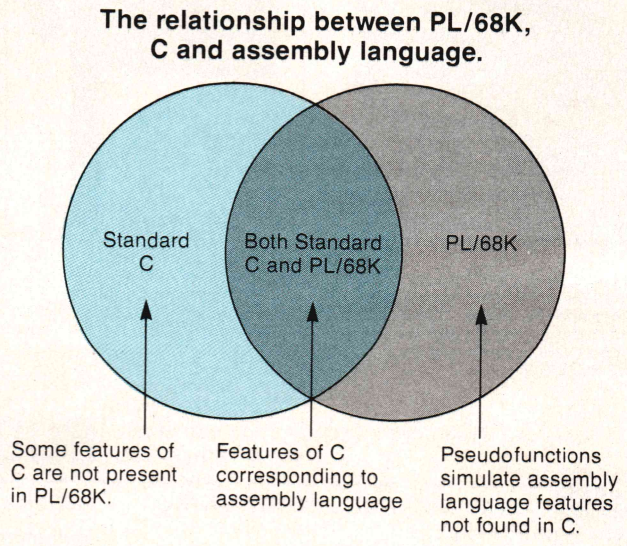


Figure 2

Note

A compiler for the PL/68K language is available from the author.

Appendix: Why Expressions Are Restricted in PL/68K

The outline of the argument is as follows. The register allocation rule conflicts with C's precedence rules (which govern the order in which operators are applied) and C's rules for converting operands from one type to another (known as the "usual arithmetic conversions" in the *C Reference Manual*). Thus, by the principle of semantic equivalence, all constructions in PL/68K that involve C's precedence and type conversion rules must be eliminated.

Consider a typical C expression, such as $a = (b * c) + (d * e);$. The register allocation rule says that this expression must be evaluated in location *a*, but that is not possible. At least two separate locations are required—one to hold the subexpression $(b * c)$ and another to hold $(d * e)$. In general, any expression involving parentheses may require more than one location to evaluate.

The only way to make a single location suffice for the evaluation of all expressions is to require left-to-right (or right-to-left) evaluation of operators. Alas, this changes the precedence of operators, except in cases such as $a = !d * c + c;$, which is evaluated left to right in C anyway.

Even such limited expressions cannot be allowed, though. C's rules for type conversions state that conversions are made during the evaluation of the right-hand side of the expression. Only after the right side is completely evaluated does the assignment take place. PL/68K has only one place to make those conversions, however—the left-hand side of the expression. The operands themselves cannot be converted because that would change their value. Thus, only one operand can be converted and only one operand can be allowed on the right side of an assignment statement.

DDJ

(Listings begin on page 101)

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 2.**

A Simple Multitasking Operating System for Real-Time Applications

by Nicholas Turner

The instruction set for the 68000 family is nearly orthogonal. This is important for a system on which a lot of assembly-language development work is to be done.

For the past year, we at Terra Nova Communications have been involved in a development project that requires a simple, fast, clean 32-bit microprocessor operating system. After a great deal of research, we were unable to find a commercial system that met our stringent requirements of extremely fast response time (even under a load of 20 users), low price (less than $10,000 for both system and software), compact code size (we wanted a system kernel, including all the utility routines discussed in this article, that required less than 20K of object code), and simple programming of applications. After some brainstorming, we created a 68000 multitasking kernel that met and even exceeded our expectations of speed and compactness. Released from hardware requirements by our decision to write the kernel ourselves, we decided to use the VMEbus hardware configuration because of its standardization, complete hardware specification, relatively low price, ease of expansion, and the availability of lots of high-speed hardware devices. We were also impressed by the reliability and ease of use of the Eurocard connectors used with the VMEbus.

Why Not Use an Existing OS?

Our requirements for speed and compactness stemmed primarily from the need to handle a large number of I/O tasks over serial lines without incurring large overheads for interrupt handling, disk access, and context switching—that is, we needed to be able to do significant amounts of I/O without slowing the system. Our kernel should eventually be able to handle 20 real-time users over serial lines at 1,200 bps, including full-speed block transfers, with no perceptible response-time delay and only minor slowing of the byte-transfer rates. We needed a system that would degrade gracefully; rather than pausing in midstream as one task takes over the system for an appreciable fraction of a second or as tasks are paged in and out, it should slow down gradually as the load increases, always providing steady and uniform output, even if it's at a reduced byte transfer rate. None of the commercial systems we examined had this property, nor were they able to handle the load required without significant degradation.

Examination of the code used in several of the commercial kernels we sampled showed some interesting reasons for this: Most kernels contained code designed to handle all sorts of unlikely circumstances that might arise in an environment where you don't know what sorts of programs might be running. Not only did this code add significantly to the size of the kernel but it also slowed down the process of context switching between tasks in many cases. We needed the fastest possible context switch in order to guarantee that minimum system time was spent on this. Fortunately, we knew exactly which applications would be running on our system, and we were able to design a complete application/system interface to make application coding easy. Because we knew that all application code running under our kernel would be "polite" (would follow all the rules of the interaction between application and kernel) and that all source code would be available for debugging, we were able to dispense with a lot of the error-handling code usually present in commercial kernels.

We discovered that another contributing factor to the time required for a context switch was the magnitude of the context that was switched. In most of the systems we examined, all the machine registers were saved and restored, and full status information was saved, both of which took up a significant amount of processing time. As we'll explain later, we fixed this in a rather unorthodox way.

Further, many commercial kernels required the use of a memory manager chip and spent significant amounts of time paging users in and

Nicholas Turner, 10 McGranahan Court, Boulder Creek, CA 95006, (408) 338-9510

out to compensate for a small system memory. We opted against memory management, mainly because it solved no problems for us. We didn't need any sort of memory protection; in fact, one of the most important criteria was that all tasks must be able to quickly read and write data belonging to any other task or to the system itself. Also, our memory requirements were not large (minimum 512K, expandable to several megabytes) and because of the amount of code sharing between tasks, the structure of our data heap, and the heap's interaction with the disk system, there was no need for hardware paging of memory.

Why Program the Kernel in Assembly?

It was clear from the start that, in order to get the kind of performance we wanted from the system, the inner kernel had to be written in native code. Even compiled C or Forth would have had to be manually "tuned" in assembly source code form. Further, we could see several difficulties with compiled language—we needed to do so many "tricky" things to extract the last few cycles from the system kernel that writing it in compiled code was out of the question. By putting the entire kernel in machine code, we simplified the effort required to make major changes (it's all in the same language) and made possible a far more complete and integrated tuning.

Eventually we expect to put up at least a C compiler and a Forth interpreter for faster development, but for the moment all development is in assembly for speed and compactness. Because assembly was the language of choice, selection of the target processor was the next important issue.

Why Use the 68000?

The eventual goal of our project is to provide a responsive telephone dial-up system that is able to support up to 30 or 40 simultaneous calls without significant performance degradation. Such a task requires a truly powerful processor, even if the whole system is written entirely in native code. For several reasons, we have chosen the 68000 family of processors for our base hardware. The most important reason is that the instruction set is extremely versatile and powerful. It is in many ways a true 32-bit instruction set, although, unless you are using a 68020, you must put up with slightly slower memory access for 32-bit reads and writes because of the 16-bit data path.

The instruction set for the 68000 family is nearly orthogonal—that is, almost every instruction can be used with any of the 12 addressing modes. This is important for a system on which a lot of assembly-language development work is to be done because the programs become much easier to generate, read, and debug. Unfortunately, even the 68000 is not perfect. Several times we've encountered annoying restrictions—for example, we've often cursed our inability to do a PC-relative store.

The 68000 also has another advantage for assembly-language programmers: The memory architecture in its native mode, without the extras added by a memory management chip, is perfectly flat. That is, the address space is completely continuous from $00 0000 all the way to $FF FFFF—or $FFFF FFFF if you have a 68020. For an assembly hacker, this is far more desirable than the segmented architecture required with the Z8000 or 80286 or with any 8- or 16-bit processor. For people writing in a high-level language, this is not an issue because they never deal directly with the memory at all. But for us, it's nice to be able to chop up that big address space in any way we like. As I'll explain later in this article, we have chosen to make it into an enormous heap and virtual disk area, thus making the fullest possible use of each and every byte.

Finally, after it became clear that the VME hardware bus was the best bet for our needs, the 68000 processor family was the logical choice because of the large number of 68000-oriented products available for the VME architecture.

In this article I'll describe briefly how our operating system fits together, and then I'll get to the fun stuff: the tricks and shortcuts we used to get such incredible performance out of the 68010 in our system. If you are already familiar with how a multitasking operating system fits together, you might want to skim down to the tricks and shortcuts.

(Continued from page 45)

Memory Structure

RAM memory is divided into two areas: the system zone and the heap, as shown in Figure 1 (page 47). The system zone begins at memory address 0 with the 68000 vector list and continues up from there. It's quite small and contains only a few data structures. Figure 2 (page 47) diagrams the structure of the system zone.

The 68000 vector list contains all the hardware vectors required for processing of interrupts and exceptions. It requires $300 bytes. The data in the vector list is first set up by the initialization section and modified thereafter by the I/O manager as device interrupts are added or deleted.

Above the vector list is a small zone containing miscellaneous system variables and pointers. The time of day and date are kept here, along with pointers for the various linked lists maintained by the kernel, plus some other information that needs to be quickly accessible via absolute short addressing (the fastest way to get at a memory location from the 68000). Several I/O devices also have data here, where it can be accessed by interrupt routines without the overhead of following pointers through memory.

Following the system variable zone, we have a rather strange beast in today's world: an old-fashioned jump table! This table contains more than 100 absolute-long address mode JMP instructions at the moment—hundreds more are planned.

The next structure in the system zone is the task control buffer (TCB) table, which is an array of data structures that are linked via pointers into several doubly linked lists. Each task is associated with one TCB in the TCB array. When a task releases control to the next task, the context switcher reads the pointer from the outgoing task's TCB to get the address of the next task's TCB. This prevents unnecessary overhead while reading TCBs belonging to inactive tasks in the array because they are not part of the active task linked list. It also makes possible an extensible TCB array: If the primary array is full when another task is about to be spawned, the task manager can allocate a nonrelo-

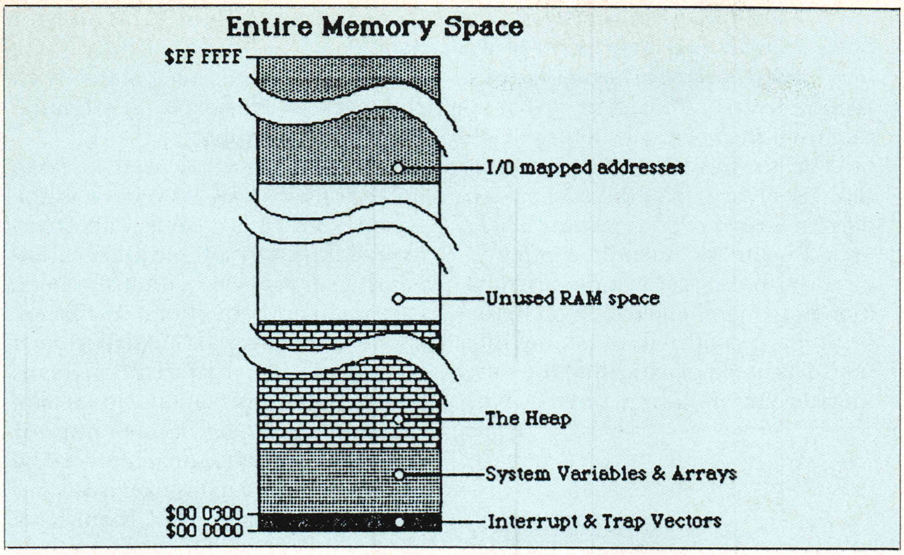


Figure 1: *Basic organization of the 68000 memory space*

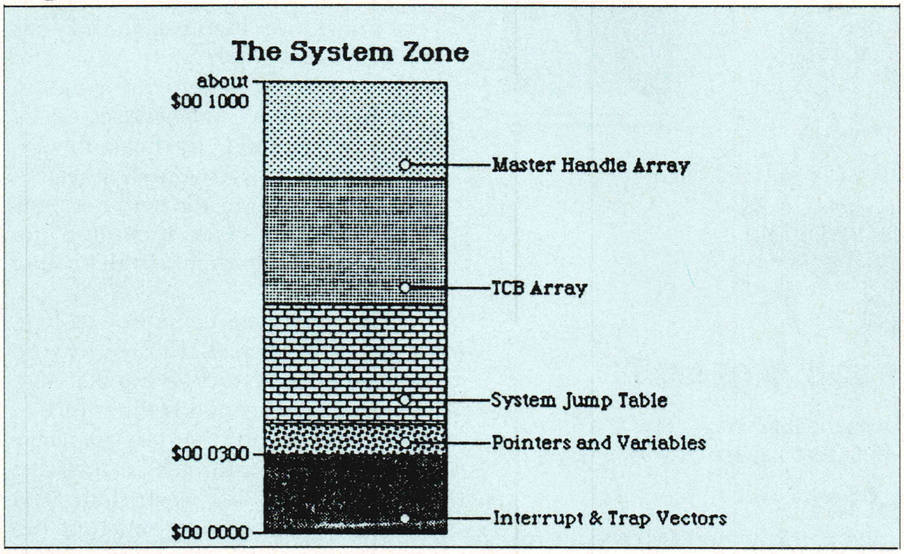


Figure 2: *The system variables and arrays*

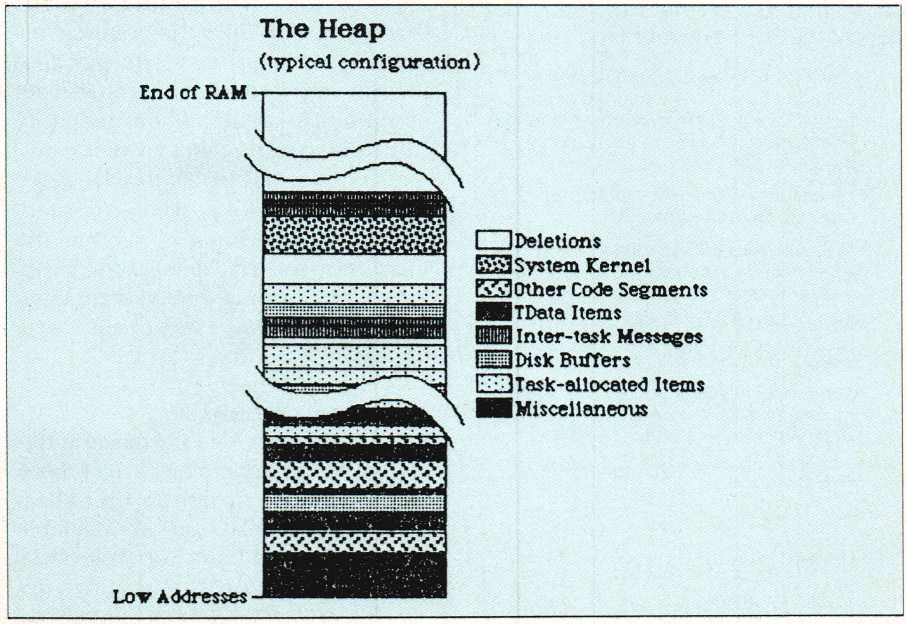


Figure 3: *Typical structure of the heap*

(Continued from page 46)

catable item in the heap and continue the TCB array there.

Another extensible array follows the TCB table: The master handle array contains handles to relocatable heap items. (A handle is the address of a pointer.) All accesses to relocatable heap items must be dereferenced (followed through the handle to the pointer to the actual heap item) before a task can use the item. That way, if an item is relocated by the heap munger (see description of the heap munger, later) during its background heap-optimization, any tasks that own the relocated item will still be able to find it because the heap munger always fixes the master handle so it is correct after moving a heap item. This master handle is often located in the master handle array, although it doesn't need to be. Wherever it is, though, it must be in a nonrelocatable place so that the heap munger can follow a pointer back from the heap item to its master handle.

After the last master pointer is an end mark. At the next 32-byte boundary, the heap begins.

The heap (Figure 3, page 47) takes up all the available RAM beyond the system zone. It is a single data structure composed of chunks called items. Every single byte of the heap belongs to an item of one sort or another. An item can be a deletion, or it can contain actual information. Items that contain information can be allocated (owned by one or more tasks) or unallocated. Unallocated items can be purged by the heap munger (see later) if it needs to make more room for a memory request from a task. For speed and ease of programming, every single heap item begins and ends on a 32-byte boundary.

Virtually all system information that doesn't have to be addressed absolutely is stored in the heap, in various heap items owned by the system kernel. In addition, the heap contains all executable code, including the system kernel itself, in chunks called code items.

Each heap item contains a 32-byte heap header record, followed by zero or more 32-byte data blocks. The header record contains the information necessary for the heap manager and the heap munger to identify, move, and validate each item. The heap is organized, like much of the rest of the system, as a doubly linked list. As the heap munger scans through it, it follows the pointers forward or backward to verify that all is in order. If it finds anything that is not completely kosher, it immediately stops the system, takes over the system console, and enters the debugger with a descriptive error message. When a bug occurs, it is often the heap munger that detects the problem (in the form of a messed-up heap header) before anything else happens.

Division of Labor

The kernel is divided into several distinct blocks of code. They are of three types: one-time routines (initialization), system calls (routines available from every task), and discrete tasks (self-contained programs that run under the context switcher, just as do application tasks). Some of the system

Call name	Description
	(Task Manager Calls)
Spawn	Create a new task
Kill	Destroy a task (by task number)
Suicide	Kill the calling task
	(Heap Manager Calls)
HeapGimme	Allocate an item in the heap
HeapDel	Release (delete) a heap item
FillZero	Re-initialize a heap item
GetMaster	Assign a master handle for an item
	(Message Manager Calls)
SendMsg	Send a copy of a block of memory to another task
Del1Msg	Delete message from top of incoming queue
DelMsgs	Delete entire incoming message queue
TxtMsg	Send a message of type "TEXT"
GetMsg	Fetch next message in queue
HandleMsg	Analyze incoming message and handle if standard type
	(Character I/O Manager Calls)
DevReq	Request a character I/O channel
DevDemand	Demand a character I/O channel (usually impolite)
DevRel	Release a character I/O channel
PrToStd	Select this task's standard character I/O device
PrToMem	Select the MemPrt device (see text)
	(Text Manager Calls)
GetCommand	Input a command line and parse it, passing control to the appropriate routine based on the command.
AddCmdTab	Add a set of commands to the existing command set
DoCommand	Parse and execute a command already stored in memory
GetPSW	Input and encrypt a password (to be compared with an encrypted password from the user file)
MoveString	Move an ASCII string
GetLine	Input a line of ASCII text from the character I/O device
PrLine	Print an ASCII string on the character I/O device
Print	Print a line of text. The line is expected to immediately follow the JSR Print instruction.
CompString	Compare two ASCII strings
	(Miscellaneous System Calls)
Random	What system would be complete without random numbers?
Sqrt	Square root of 32-bit integer

***Table 1:** Some of Terra Nova's system calls*

TRAP-oriented calls

Instruction		Cycles used	Description
TRAP	#n	38	call the system routine
MOVE.L	2(SP),A0	16	point to word argument
MOVE.W	(A0)+,D0	8	fetch the argument
MOVE.L	A0,2(SP)	16	update return address
(variable)		(variable)	decode argument word
------			useful code
RTE		24	return to caller
		----------	
		104 (+ decode)	total cycles for overhead

JSR-oriented calls

Instruction		Cycles used	Description
JSR	Label.W	18	call low memory entry point
JMP	Label.L	12	call actual routine
------			useful code
RTS		16	return to caller

		46	total cycles for overhead

***Table 2:** Comparison of TRAP- and JSR-oriented system calls*

(Continued from page 48)

calls we've developed are listed in Table 1 (page 49). This is not intended to be a complete list, only to give some of the system's flavor.

The initialization section is the block of code that gains control before anything else happens. It starts with a brute-force approach: It grabs control from whatever operating system invokes it, and then it sets up the bare-bones data structures for the system. I'll go into greater detail about the initialization section later, when I talk about tricks and shortcuts.

The most important low-level code segment is the context switcher, which is the routine that receives control from one task and passes it on to the next. It is extremely small and extremely fast, and it makes a lot of assumptions about the tasks as it does its job. This is by design: By making assumptions and forcing the tasks to adhere to them, a lot of overhead is eliminated. Again, I'll go into detail about the context switcher in the section on tricks and shortcuts.

The task manager is composed of a group of system routines available to every task. They allow a task to create, destroy, and manipulate other tasks or itself.

The heap manager is a collection of routines that allow any task to request, release, lock, enlarge, or otherwise manipulate heap items.

The message manager is a collection of system routines that make possible a clean and well-defined message-passing protocol between tasks. Simply by pointing at a block of memory and calling a system routine, any task can send a copy of any piece of data to any other task. Reading queued messages from other tasks is similarly easy.

The character I/O manager is a set of system calls and interrupt service routines. Together with the text manager, it makes possible a simple I/O structure, in which each task can select any physical or logical device for I/O by passing a device number. Serial input is interrupt-driven, and serial output is polled. Device drivers can be added or removed from the I/O manager with another set of system calls.

The text manager is a collection of

system routines that ease the processes involved in talking to humans. It includes powerful routines to get and parse command strings as well as text-manipulation routines such as case conversion, context-sensitive string comparisons, and so forth. The powerful parsing calls make it easy to create a tiny machine-language task that includes a complete command interpreter and syntax error handler. This is very important if a significant amount of development is to be done in assembly language.

The trap manager handles all system traps except I/O interrupts, which go directly to the I/O manager. Error traps always cause the system to come to a complete standstill. This is important to us because of the close interaction between the tasks, which must always be in intimate communication to fulfill the purpose of the system. When an error trap occurs, all task switching and interrupt processing stop and the task in which the error occurred takes over the system console and enters the debugger. The human in charge can then take corrective action and restart the system with minimal damage. Obviously this approach would be completely unacceptable in a commercial operating system, but for us, it is ideal because we have all the source code for the entire system and can often correct bugs as soon as they occur.

No assembly language development system is complete without a debugger, of course, especially a multitasking one. Our debugger is a command parser that any task can invoke, either automatically (in response to an error trap) or directly. It is capable of running alongside other active tasks, even multiple copies of itself, and it allows the user full manipulation and examination of memory.

The heap munger is a distinct task, always present, always active, whose original job was to survey the contents of the heap continuously and maintain it as an efficient data structure (by using a background task for this, we avoided many complexities). The heap munger has turned into quite a bit more than just a trash compactor, however. Its responsibilities are many and varied, from checking the TCB array for integrity and waking up sleeping tasks when their ships come in to responding to messages from other tasks that want to know what the system loading is so that they can adjust their own CPU usage to increase the overall performance of the system. In fact, the heap munger is also capable of responding to text messages sent to it by a human who is operating a user task, in which case it responds by sending a plain English message back to the source task, which then displays it for the human to read. As a general rule, the heap munger performs any systemwide activity that must be performed at frequent, regular intervals. With all these responsibilities, it is the largest single code segment in our kernel, weighing in at about $A00 bytes.

The disk munger, like the heap munger, is a distinct task that is always running except when it's waiting for an I/O completion. Because its structure and function are application specific, I won't go into it in great detail here. I would like to point out, however, that by allocating a single task to handle each disk device, a large number of problems related to data contention between tasks can be avoided. The disk munger is entirely message-driven: As each task requires a disk access, it sends a message to the disk munger for the device it wants to access. Each I/O request gets added to a queue of such

MULTITASKING OS
(Continued from page 51)

messages. When the disk I/O is completed, the disk munger sets the "wake-up" flag for the requesting task, and the heap munger wakes it up on its next pass. The requesting task then looks in its TCB for the completion code from the disk munger. Because the I/O requests are high-level calls, usually implicitly including the open, read/write, and close of the file, there are few file contention problems.

On start-up, a discrete task called the system console manager initially gains control of the system console and puts up a system monitor prompt. Until another user task is spawned, it is the only task that has access to a character I/O device (and thus, to a human). The system console manager can then give way to any other user program.

Tasks

Everything that gets done on the system, with the single exception of the context switch between tasks, is done by a task. Three tasks are always present in the system: the system console task, which initially runs the system console manager; the heap munger; and at least one disk munger. The disk munger is actually optional, although it's hard for me to imagine doing any useful work without using a disk.

Each task, whether it's active or not, possesses exactly one task data item (TData item) in the heap. The TData item is crucially important to the proper functioning of the system because it's where each task stores its most important local information. Every task's TCB entry contains the master pointer to its TData area in the heap. During normal execution of a task, the 68000 register A5 always points to the base of the TData item. Before the context switcher passes control to a task, it always sets up register A5 for the incoming task.

The TData item is also the location of the task's local data stack: The stack goes from the top of the TData item down, and the task's local data goes up from the bottom. Note that it is the responsibility of each task to ensure that its stack and TData areas do not collide.

(Continued from page 52)

Each task's TData item contains spaces for various pointers and vectors associated with its current character I/O device, if any. The vectors include various standardized routines such as InCheck, which checks to see if a character is available; InWait, which waits for a character and returns with it; OutCheck; OutWait; plus several others. The pointers include the addresses of the destination for the next input byte (if any), the source of the next output byte, and so forth.

Character input is generally interrupt-driven. Because a task that is awaiting input is often "asleep" (not in the active TCB list), it is necessary for the interrupt routine to set a flag that tells the system to wake up the owner of the device. Then, after the interrupt has been handled, the heap munger, which is always awake and active, catches the set flag the next time it gets control and performs the actual manipulations to return the task's TCB to the active list. In order to provide an input time-out, the heap munger also awakens each task once every ten seconds so that the task can test for the time-out.

Each task may request to be assigned to a character I/O device. If a device that is requested is currently assigned to another task, the requestor will usually have to wait until the device is available. For special cases such as error traps, however, there is a system call that allows the task to demand to be connected to a device. When a demanded device is released, it reverts to the task (if any) to which it was attached originally. When a requested device is released, it always becomes available (unattached) again. Some devices can be attached to multiple tasks at the same time—for example, there is a device called MemPrt that reads and writes to memory as if it were a character stream from/to a serial device. This "virtual" device can be simultaneously active for different tasks, each with its own set of pointers in the TData item for reading and writing.

I won't go into great detail about our I/O drivers or the low-level structure of the I/O routines. This sort of information is readily available in several forms in computer bookstores.

The Good Stuff: Tricks and Shortcuts

The first thing our operating system does when it gains control of the processor is to deviously remove the existing operating system. This it does by using a trick to get into supervisor mode (see the listing, page 102). First, it shoves a new address into the privilege exception vector, which is in the hardware vector table in low memory. This address happens to be that of the second instruction following the one that does the store to the vector. The next instruction is a privileged but otherwise harmless one. Thus, when it tries to execute it, it traps to the privilege exception vector and thence to the next instruction in the program. If through some quirk we happen to be in privileged mode already, the processor harmlessly executes the privileged instruction and falls through. Now we are in privileged mode, and we can quickly grab the rest of the system.

Next, we turn off all the interrupts in the system as quickly as possible. It is important to do this before clearing memory because a stray interrupt might happen before we can turn it off and it must still vector properly. After all interrupts are out of commission, we copy our own set of vectors into the interrupt table.

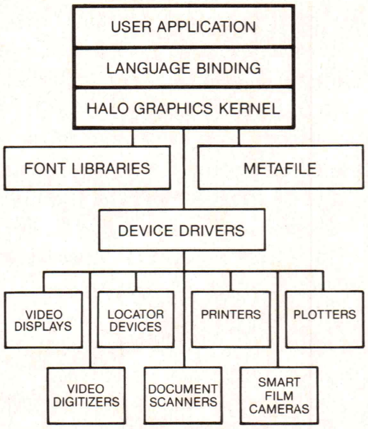
The jump table (discussed later) is next moved into place in low memory. It is read from a section of object code within another assembled module. It's nothing more than a sequence of more than a hundred absolute-long JMP instructions.

Next, we clear the rest of memory—except the kernel, of course—to zeroes. This is both a general safety measure and an aid in debugging: If a chunk of memory is nonzero, we can be certain that something we did caused it to be that way. Also, it's nice to be able to assume that unused memory is always zeroed out; it makes for much faster initializations later on, once the system is running.

The system zone requires a certain amount of initialization. The task control blocks must be set up and the end marks for the TCB and master

HALO the market standard for graphics

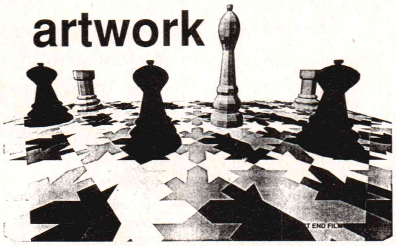
HALO provides software implementors with a complete graphics development environment.



Device Independent

Each copy of HALO comes with a full complement of display, input, printer, film recorder and plotter device drivers. Metafile creation and display are included. Media Cybernetics is continually developing new device drivers to meet the ever changing new hardware technologies.

HALO supports all IBM compatible MS/PC DOS systems including Texas Instruments Professional™, AT&T 6300™, IBM™ 3270-PC and of course the IBM PC, XT and AT. Over twenty graphics display boards manufactured by IBM and leading third party vendors are supported. A single device driver supports all display modes on each device. Image input from video digitizers and document scanners is supported. New device drivers include the IBM EGA and Professional and the AT&T Image Capture and Display boards.



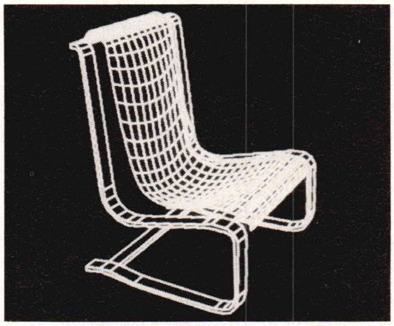
This picture was created with "Artwork", a HALO-based application written by West End Film of Washington, DC.

De-Facto Standard

First released in 1982, HALO has an established base of over 40,000 end users and over 100 corporate clients. Numerous HALO-based CAD, solids modeling, presentation graphics, art packages, mapping and other vertical market software applications are commercially available.

As early as June 1984, PC World featured an article entitled, "HALO A new software library leads the way toward graphics standardization and portability." The July 9, 1985 issue of PC Week" featured users stories from both Rockwell International Corp., and Lawrence Livermore Labs on how they use HALO to save development time and money. Most recently, Mini-Micro Systems August 1985 issue stated,'"Widely used, HALO has attained de facto-standard status. It's certainly the most widely used library." HALO has achieved this status because it provides a complete device independent graphics environment for software developers. Since the HALO interface rarely changes, compatibility with a new device is achieved simply by adding a new device driver.

By incorporating all the HALO device drivers into their applications, licensed commercial software developers solve distribution and compatibility problems.



This picture was created with "Design Board 3-D", a HALO-based application written by MEGA-CADD of Seattle, Washington.

Powerful Graphics Toolbox

HALO has over 170 functions. Standard functions include: point, line, arc, circle, ellipse, polygon, etc. Coloring control functions are available for color selection, palette management, dithering, and textures.

Advanced features encompass: image compression routines for reducing data storage requirements; both absolute and relative addressing plus individual cursors for graphics text and interaction; world and normalized coordinate systems for windowing, viewport and clipping; and a metafile reader and generator.

HALO's raster extension supports rubberbanding and non-destructive modes so that polygons and text may be interactively created and moved over a displayed image.

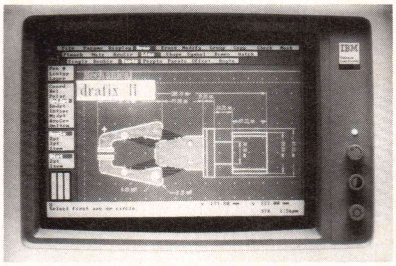
There are currently over twenty bit mapped and stroke text fonts available for HALO. The object oriented type faces provide user definition of line width, size, proportional width, angle, and interior fill style.

HALO's Virtual Rasterization Interface (VRI) frees users from the limited resolution of their graphics board by modeling the graphics display in memory and having the hardcopy produced in the maximum resolution of the output device.

HALO can be addressed as a linkable library or as a resident driver depending on the application.

Faster than VDI

In benchmark tests by independent companies evaluating HALO and GKS, HALO was determined to be three to ten times faster depending on the function tested. These benchmark results made HALO a requirement for many government agencies and large corporations. This gives HALO compatible hardware and software significant inroads into these agencies and corporations.



This picture was created with "drafix II", a HALO-based application written by FORESIGHT Resources Corp. of Overland Park, Kansas.

Easy to Learn

HALO comes with a computer aided tutorial. This tutorial, entitled LEARNHALO, teaches users to become proficient graphics developers in less than a day by providing step-by-step instructions through graded examples.

HALO is written in Assembler and callable from the language you are already comfortable with: Interpretive and Compiled Basic, Professional Fortran, MS Fortran™, Pascal, Turbo Pascal™, MS C™, Mark Williams C™, Lattice C™, Aztec C™, Computer Innovations C86™, and Golden Common Lisp™.

Increase Market Value of Hardware

Graphics hardware manufactures receive distinct marketing advantages by having their products supported by HALO. HALO is the gateway to an extensive body of commercially available software.

Pricing Information

A single copy of HALO for one language costs $250.00. A second language binding ordered at the same time is an additional $150.00. Author, Corporate and Site licensing and OEM agreements are negotiated on a case by case basis.

To order, or receive more information on HALO and other Media Cybernetics, Inc., products contact:

media cybernetics, inc.
7050 Carroll Avenue
Takoma Park, MD 20912
(800) 446-HALO or (301) 270-0240
Telex #332014

HALO, LEARNHALO, Texas Instruments Professional, AT&T, IBM, MS Fortran, Microsoft C, Turbo Pascal, Mark Williams C, Lattice C, Aztec C, Computer Innovations C86, Golden Common Lisp and GKS VDI are all registered trademarks of Media Cybernetics Inc., International Business Machines, Texas Instruments Inc., AT&T, Microsoft Corp., Borland Intl., Mark Williams Co., Lattice Inc., Manx Software Systems, Computer Innovations Inc., Gold Hill Computers, and Graphics Software Systems Inc., respectively.

Circle **no. 199** on reader service card.

(Continued from page 54)

handle arrays must be set in place. The values in the miscellaneous system data area must also be initialized.

After the system zone is in place, the heap is defined from the next 32-byte boundary to the end of memory. Three heap items are initially set aside: a deletion from the beginning of the heap to the start of the kernel's code, a fixed (immovable) code item for the kernel, and another deletion from the end of the kernel to the end of memory. Soon, the other tasks will be carving up the big deletions for their own use.

Each time a task is spawned, the task manager creates a new TCB and a new TData item. The first task to be spawned is the system console manager. The system task manager allocates an item of the proper size from the heap, creates a TCB for the new task, and adds the TCB to the current linked list of active tasks' TCBs.

After the system console task is spawned, the heap munger and disk munger are also spawned. Note that none of them begins to execute until the initialization code jumps into the middle of the context switcher.

We discovered that one of the biggest sources of "extra" system overhead in commercial operating systems is the need to manage tasks that might possibly get out of hand and take over the system. In most cases, such "runaway" tasks are avoided by using a hardware timer to assure that any given task will only be able to run for a preset time. If a task spends too much time without releasing to the system, the timer interrupt occurs and vectors the CPU through to the supervisor program. This program then saves the existing task's registers and status and restores those of the next task in line, which then is off and running with a newly reset timer. All of this requires a lot of tricky and complex code to ensure that no task can "run away" and lock up the system. Also, the constant saving and restoring of registers and status, which is necessary because the context switch is interrupt-driven, adds measurably to the overhead required for a context switch.

We debated for some time about the best way to reduce this overhead.

Finally, we settled on the answer: We developed our kernel as a nonpreemptive task controller. This means that there is no hardware timer to interrupt each task after some preset interval, and no routines are required to service such an interrupt. Instead, we use a simple context switcher, one that doesn't even bother to save registers or the previous task's status bits. It is the task's responsibility to call the context switcher often enough to ensure smooth system operation and to make sure that it saves any registers that it needs (other than A5 and the stack pointer). For our application, this is perfect because most of our tasks spend most of their time waiting for input or output in the inactive task list, during which time the other tasks can run unhindered.

When a task has finished with the CPU and is ready to let the next task in the active list run for a while, it simply calls the context switcher as a subroutine using a JSR instruction to a low-memory JMP instruction whose address is fixed regardless of the location of the context switcher. Even with the added overhead of the extra JMP instruction, this method of calling is considerably faster than a TRAP instruction—the usual method of calling a context switcher.

Once the switcher has control, it checks to see if the system tasking is stopped (see the listing). If it is, then the calling task immediately gets back control through a simple RTS. Otherwise, it gets ready to call the next task.

The address of a task's TCB is in its TData area. This address is loaded into A0. Then the current TData base address (in register A5) is subtracted from the stack pointer, yielding a relative displacement, which is stored in the TCB. Now we're ready to move on to the next task.

The address of the next task in the circular linked list of active tasks is fetched from the old task's TCB into A0. Register A5 is set to point to the new task's TData area by moving its address from the TCB, and the stack pointer is restored from the relative displacement by adding A5 to it. Then a simple RTS returns control to the task.

This otherwise trivial scheme has one slight complication: Frequently, a task will release control with the intention of going to sleep for a while. This happens, for example, when the RAM buffer has no input characters for the character device attached to a task that is waiting for input. When the input finally happens, the interrupt routine sets a flag that causes the heap munger to wake up the task when it next checks for such a situation. The problem is that the context switcher I have just described has no provision for putting the old task to sleep: It assumes that both tasks want to stay awake.

So, we created an alternate context switcher that removes the outgoing task from the active list before calling the new one (see the listing). When a task wishes to go to sleep, it simply calls the alternate context switcher. The primary difference is that before it moves on to the next task, the alternate switcher does some standard and fast list manipulation.

Most commercial operating systems use one or more of the TRAP instructions to perform system calls, usually going on the premise that they are there for that purpose and that the TRAP instructions allow programs to be more general and more easily relocated. Unfortunately, TRAP instructions on the 68000 cause a ma-

(Continued from page 57)

jor problem: They take a long time to execute, as do the instructions to decode their arguments. The RTE instruction, which returns from a TRAP, shares the same problem.

We could understand the importance of relocatable programs, certainly, but we felt the long-lasting TRAP instruction was too much to ask of an ultra-fast, real-time system. We therefore designed a different, faster way to call system routines: the JSR instruction and an old-fashioned jump table in low memory (see Table 2, page 49, for a timing comparison). Instead of using a TRAP instruction with an argument following in the next word, we simply call one of many entry points that are at absolute locations in low memory. Each entry point consists of a single JMP instruction to the actual system call entry point. Not only do we save the extra time required for the TRAP and RTE instructions but we also avoid having to extract the argument word from the bytes following the TRAP instruction and having to add two to the return address to jump around the argument because the argument is implicit in our choice of which routine to call. (With TRAP-based system calls, an argument is required to specify which system call to use because there are only 16 traps. With JSR calls, there can be hundreds of separate entries, so no argument is required to specify which call is to be used.) By using subroutines instead of traps, we shaved more than 100 machine cycles from every single system call, which makes a measurable difference in a machine that uses lots of system calls.

Conclusion

With the two context switchers just described, a small set of carefully designed system routines, a somewhat unusual system calling procedure, and a certain amount of cooperation from the application programs, we have vastly increased the throughput of our system. Our approach is obviously not well suited to most projects as it requires a considerable amount of skill and cooperation on the part of the programmers. Furthermore, because of its nonstandard nature, it is poorly suited to any applications that are written for commercial systems—at least until we get a C compiler running! If you need an extremely fast multitasking system for a specialized real-time application and are strapped for funds, however, this approach can turn a relatively inexpensive microcomputer into an amazingly powerful system. To date we have done just that for four different hardware configurations.

I'd be happy to discuss the philosophy and tricks in greater detail with anyone who is interested. For more information on our operating system, write to Nick Turner at the following address:

Terra Nova Communications
10 McGranahan Ct.
Boulder Creek, CA 95006

Or call me at (408) 338-9510. Terra Nova Communications is a consulting firm specializing in small multitasking systems. **DDJ**

(Listing begins on page 102)

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 3.**

C+

The Best C Book
A Powerful C Compiler
One Great C Value $39.95

A good C book just isn't complete without a good C compiler to go with it. That's why we give you both. You get a comprehensive 450 page book and a full feature standard K&R C compiler with the Unix V7 Extensions. The Book is loaded with examples that teach you how to program in C. And our fast one pass C compiler comes with an equally fast linker so you don't waste a lot of time watching your disk drives spin. You also get a Unix compatible function library that contains more than 150 functions (C source code included). And if all that isn't enough, we offer you a 30 day money back guarantee. So what are you waiting for? The exciting world of C is just one free phone call away.

Language Features

- Data Types: char, short, int, unsigned, long, float, double
- Data Classes: auto, extern, static, register
- Typedef, Struct, Union, Bit Fields, Enumerations
- Structure Assignment, Passing/Returning Structures

Functions

abs	conbuf	feof	getcseg	isascii	movmem	replace	strcat
asm	conc	ferror	getdseg	iscntrl	open	repmem	strcmp
asmx	cos	fflush	getd	isdigit	outp	rewind	strcpy
atan	cpystr	fgets	putd	islower	peek	right$	strlen
atof	creat	fileno	getdate	isprint	perror	rindex	strncat
atoi	cursblk	filetrap	gettime	ispunct	poke	rmdir	strncmp
atol	curslin	find	geti	isspace	poscurs	scanf	strncpy
bdos	curscol	floor	puti	isupper	pow	setbuf	strsave
bdosx	cursrow	fopen	getkey	itoa	printf	setbufsiz	system
bios	cursoff	fprintf	getmode	keypress	putc	setcolor	tolower
biosx	curson	fputs	setmode	left$	putchar	setdate	toupper
calloc	delete	fread	gets	len	puts	settime	ungetc
ceil	drand	free	getw	log	putw	setjmp	ungetch
cfree	exec	freopen	heapsiz	log10	rand	setmem	unlink
chain	execl	fscanf	heaptrap	longjmp	read	sin	write
character	execv	fseek	hypot	lseek	readattr	sound	writechs
chdir	exit	ftell	index	malloc	reach	sprintf	xmembeg
chmod	exitmsg	fwrite	inp	alloc	writech	sqrt	xmemend
clearerr	exp	getc	insert	mathtrap	readdot	srand	xmemget
close	fabs	getch	iofilter	mid$	writedot	sscanf	xmemput
clrscrn	fclose	putch	isalnum	mkdir	realloc	stacksiz	xmovmem
cmpstr	fdopen	getchar	isalpha	modf	rename	str$	_exit

MIX Editor
$29.95

When you're programming in a high level language you need a high powered editor. That's why we created a programmable full/split screen text processor. It lets you split the screen horizontally or vertically and edit two files at once. You can move text back and forth between two windows. You can also create your own macro commands from an assortment of over 100 predefined commands. The editor comes configured so that it works just like Wordstar but you can change it if you prefer a different keyboard layout. The editor is a great companion to our C compiler. Because they work so well together we want you to have both. To make sure you do, we're offering the editor for just $15 when purchased with the C compiler.

ASM Utility
$10

The ASM utility disk allows you to link object files created by Microsoft's MASM or M80 assemblers. Lots of useful assembly language functions are included as examples.

ORDERS ONLY
1-800-523-9520
IN TEXAS
1-800-622-4070

Canadian Distributor
Saraguay Software: 416-923-1500

NOT COPY PROTECTED

Editor $ ________ (29.95)

C $ ________ (39.95)

C & Editor $ ________ (54.95)

ASM Utility $ ________ (10.00)

TX Residents $ ________ (6.125% sales tax)

Shipping $ ________ (see below)

Total $ ________

☐ Check ☐ Money Order

☐ MC/Visa# ________________ Exp ________

Shipping Charges: (No charge for ASM Utility)
USA: $5/Order
Canada: $10/Order
Overseas: $10/Editor • $20/C • $30/C & Editor

☐ PCDOS/MSDOS (2.0 or later)
- ☐ IBM PC Single Side
- ☐ IBM PC Double Side
- ☐ Tandy 2000
- ☐ 8 Inch
- ☐ Other ________

☐ CPM 80 (2.2 or later)
- ☐ 8 Inch
- ☐ Kaypro II
- ☐ Kaypro 4
- ☐ Apple (Z80)
- ☐ Osborne I SD
- ☐ Osborne I DD
- ☐ Morrow MD II
- ☐ Other ________

Name ________________

Street ________________

City ________________

State ________________

Zip ________________

Country ________________

Phone ________________

MIX software
2116 E. Arapaho
Suite 363
Richardson, TX 75081
(214) 783-6001

Ask about our volume discounts.

DI

Bringing Up the 68000—A First Step

by Alan D. Wilcox

Adapted by permission of the publisher, Prentice-Hall Inc., from the forthcoming book Designing and Troubleshooting a 68000 Microcomputer System *by Alan D. Wilcox, available 1986. No part of this adaptation may be reproduced, in any form or by any means, without written permission from the publisher.*

The impact of the free-running technique on hardware development is startling. The 68000 kernel can be made to run so easily that a logic probe can test it.

There are two ways to bring up a new 68000 microcomputer system: the hard way and the easy way. The hard way is to use the traditional approach of designing the hardware and then using a development system along with test software and some in-circuit emulation. Given enough hours of testing and correcting problems, the 68000 system has a good chance of running successfully. In contrast, the easy way is to design, build, and test the hardware module by module using the 68000 as a free-running processor.

The impact of the free-running technique on hardware development is quite startling. The 68000 kernel shown in Figure 1 (page 61) can be made to run so easily that a logic probe can test it. You don't need to use sophisticated digital development tools such as a logic analyzer or a development system with an in-circuit emulator. If troubleshooting is necessary, you need only a common dual-trace oscilloscope.

Free running the 68000 means that the processor is allowed to execute a do-nothing instruction continually. This is accomplished by breaking the normally closed loop between the 68000 and its memory, as shown in Figure 2 (page 61). Instead of carrying program instructions from memory, a one-word instruction (call it a NIL instruction) can be jammed onto the data bus. (The mnemonic NIL is not part of the 68000 instruction set per se; I coined it as a simple expression of the instruction used for free running.) Thus, when the 68000 reads the data bus for an instruction, it fetches the NIL word, executes it, increments the address, and reads the next NIL. This cycling repeats over the entire 16-megabyte address range; when the processor reaches the end of the 16 megabytes, it simply starts over again.

The strategy for bringing up the 68000 is to design, build, and test the 68000 kernel shown in Figure 2. Next, design and build one additional module, connect it to the kernel, and test it while the processor is free running. Add yet another module and test it while free running. You can free run the 68000 all the way through the construction of a complete CPU board. In fact, if a processor board fails, you can usually free run it to help speed troubleshooting. The only part of the system that you cannot test easily while it is free running is the data bus itself, because the NIL instruction is forced on the bus.

The Steps

Several steps are involved in bringing up the 68000 using the free-running technique. The intent of this article is to describe the necessary first step: how to get the kernel running. Once you have the kernel in operation, the rest is fairly straightforward. Here is a brief overview of the entire scenario to complete a working 68000 CPU board:

1. Bring up the kernel. Design, build, and test the power system, the 68000 clock and drivers, the reset and halt module, and the 68000 module.
2. Add a wait state and data transfer acknowledge (DTACK*) generator module.
3. Add RAM and EPROM decoding circuits, connect address and control bus circuits.
4. Write a simple looping program for a pair of EPROMs. Remove the NIL instruction and close the broken loop between the EPROMs and the 68000 data bus. The processor should now be able to read its stack and program counter vectors from the EPROMs and execute the loop program.
5. Add the RAM connections to the data bus. If the 68000 is still running successfully with the simple loop program, add more code to include reading and writing RAM. If the code is a very tight loop, an oscilloscope will synchronize easily and you can use it to check the timing of the various control lines to all the memory in the system so far.

Alan D. Wilcox, 60 South Eighth St., Lewisburg PA 17837

6. Modify the reset and EPROM-control circuits so that the EPROMs do not have to be decoded at address 0 except during reset. Normally, the low memory addresses should be RAM so that exception vectors can be altered dynamically; EPROMs should be elsewhere.

7. If at least 4K of RAM has been decoded starting at address 8, and the EPROMs are decoded for 0 to 7 and $8000 to $BFFF, then the system can use the Motorola TUTOR EPROM set. When you restart the system, assuming that it does not halt, you can use an oscilloscope to see the activity on the various processor lines while the monitor waits for a console keystroke.

8. Add a 6850 ACIA decoded at $010040 to serve as a console port and test it with the TUTOR EPROM set. Run various memory-testing commands and scope loops to check operation of the new system.

The First Step

As stated earlier, the first step is to bring up the kernel in the free-running mode. It seems a bit overwhelming when you first try doing it, but it really is quite simple. Unless you have made a wiring error, the 68000 is virtually guaranteed to come alive and begin executing the NIL instruction. To bring up the kernel, you need the power system, the clock and drivers, the reset and halt module, and the 68000 module.

The size of the power supply depends on the nature of the system and what loading it will have in the final configuration. In my own case, I intended to use the 68000 processor board in an IEEE-696 (S-100) system, so I needed on-card regulation from an 8-volt system supply. A common 7805 circuit was adequate for the processor and its RAM and EPROMs; I used a second 7805 circuit for the rest of the LS-TTL logic on the CPU board.

Watch the Motorola data manual for footnotes. In the case of the 68000, although the data indicates a power requirement of 300 mA or so, that is not the whole story. The fine print at the bottom of one page casually mentions that the 68000 might require a peak current of some 1.5 A. Make sure the power supply can handle the peak current without falling out of regulation. Likewise, power and

Address	Data	Program
00 0000	0000 0000	ORI.B #0,D0
00 0004	0000 0000	ORI.B #0,D0
00 0008	0000 0000	ORI.B #0,D0
00 000C	0000 0000	ORI.B #0,D0
00 0010	0000 0000	ORI.B #0,D0
.		
.		
.		
FF FFFC	0000 0000	ORI.B #0,D0.

Table 1

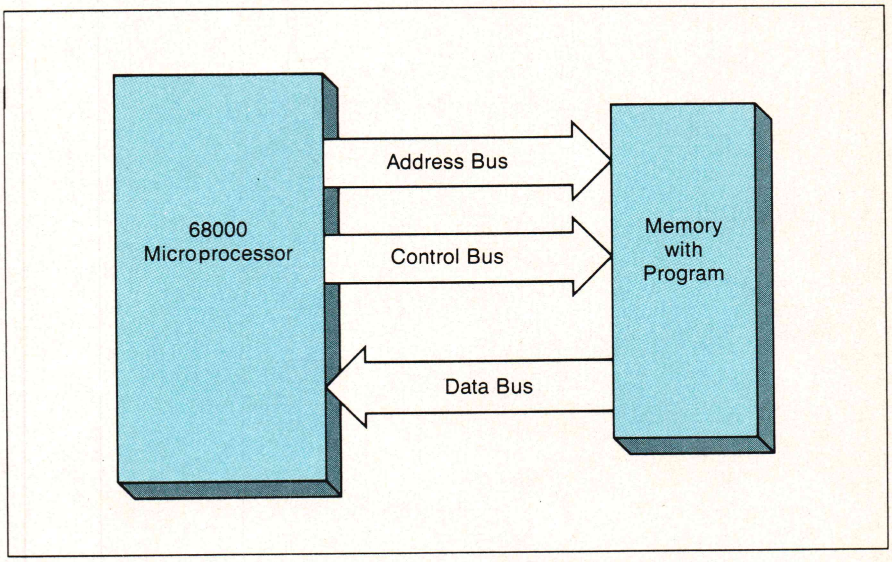


Figure 1: *The 68000 kernel is the essential hardware for program execution.*

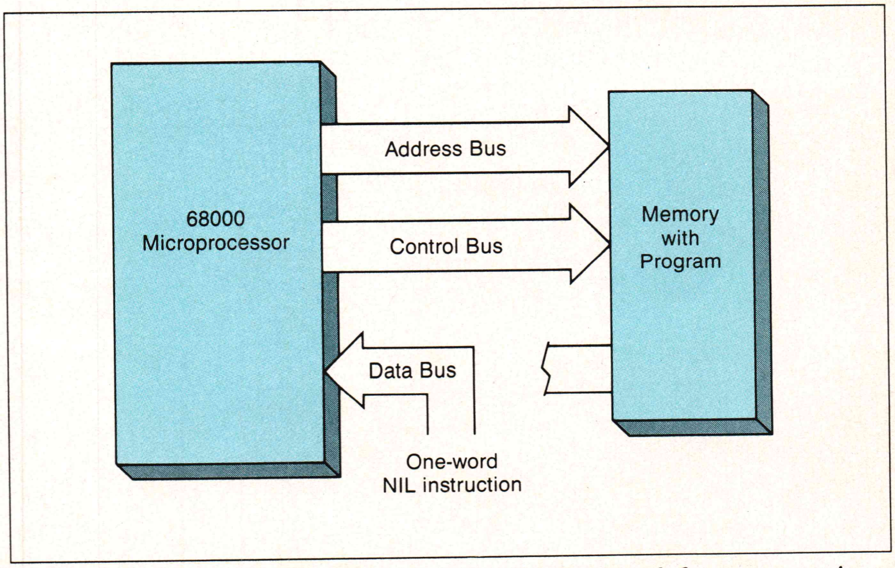


Figure 2: *To free run the 68000, the normal feedback path from memory is disconnected, and a NIL (do-nothing) instruction is substituted.*

68000
(Continued from page 61)

ground leads to the 68000 need to be heavy, say #24 wire rather than #30 wire-wrap wires. Locate bypass capacitors close to each of the power and ground connections.

You can design and build the clock circuit using a crystal, some resistors and capacitors, and a 7404 or similar; doing this is hardly worth the effort, though. For prototype work, being able to change the clock frequency easily without redesigning the circuit is a distinct advantage, so using a DIP oscillator is appropriate. I used a 6-MHz oscillator in my S-100 prototype to keep within the bus specification; only after I had finished the system did I run it up to 10 MHz and later to 12 MHz.

Also, you will use both the clock signal and its complement in the final circuit design. The complement clock could be derived from a 74LS04, but that would introduce a skew be-

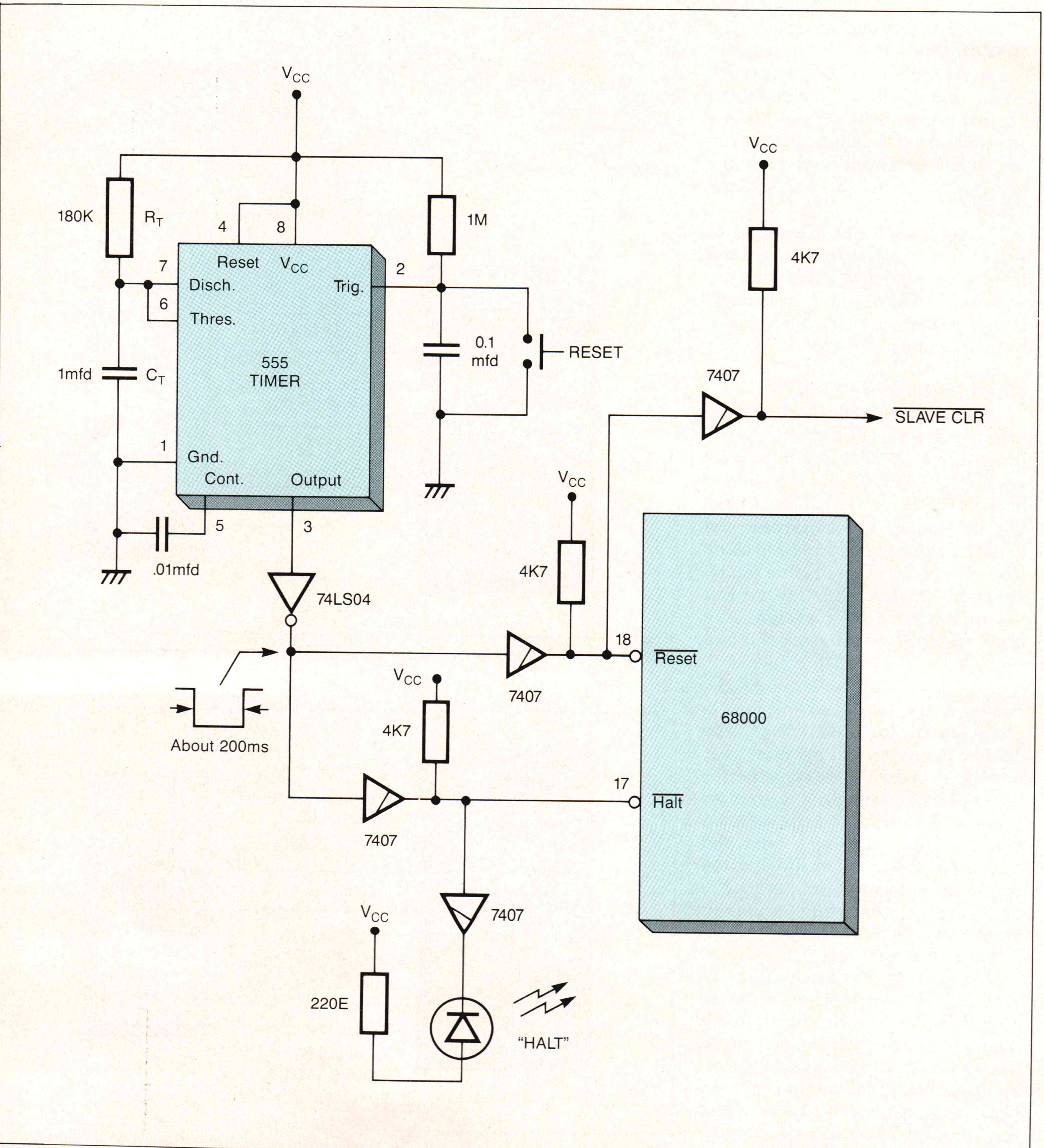
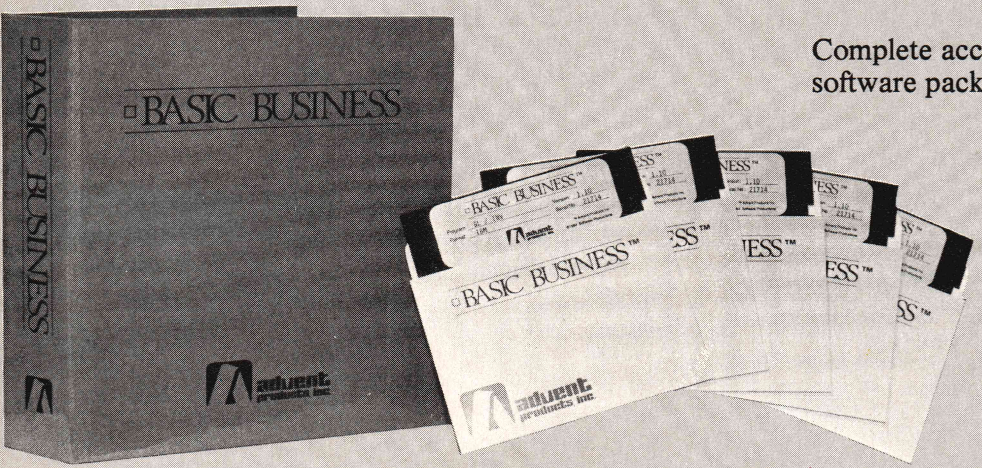


Figure 3: *Circuit diagram of a simple power-up and reset timer circuit for a 68000 processor. Note the use of open-collector devices on the bidirectional HALT* and RESET* controls of the 68000.*

TIMELY REPORTS, ACCURATE RECORDS, RELIABLE INFORMATION......



Complete accounting software package, only **$89.95**

......this is □ BASIC BUSINESS

Price is nice - and so is a long list of features - but when you choose a software package to automate your accounting, don't lose sight of the basics of good business record keeping. Basic Business, an all-in-one accounting software package consisting of General Ledger, Accounts Receivable, Accounts Payable, Payroll, Inventory Control, Sales Order Processing, and Purchase Order Processing. It offers an excellent price and extensive feature list without sacrificing these basic business principles:

Full Audit Trails - to give you complete confidence in the accuracy of your data and provide crucial backup information when you need it. And double entry accounting keeps your books in balance.

Complete Integration - provides efficient processing for all your business transactions, updating all of your accounting records instantaneously. You only enter the transaction once, and all supporting modules are updated automatically.

Flexibility - Basic Business can be adapted to your way of doing business, including balance forward or open item accounts receivable, departmental or consolidated general ledger, variable aging periods and easy entry of manually written checks and voids.

Extensive Reporting - each accounting module provides complete reports, including master lists, transactions, journals, statements and forms. In addition, all data files are compatible with Ashton-Tate's dBase II and dBase III, for the ultimate in custom reporting capability.

With some accounting software, even packages costing hundreds of dollars more, you must make detailed estimates and complex calculations for the maximum number of customers, transactions, inventory items, etc., before using the system. Then, when your business grows to exceed these original estimates, you must repeat this process.

Not with Basic Business! All data files are automatically initialized when you install the system on your computer. Files can grow dynamically as your business increases and areusually limited only by the amount of you exceed your original estimates.

FORMS ARE NO PROBLEM! Basic Business uses standard forms for invoices, statements, checks, purchase orders, etc., which may be ordered with your company name, adddress and logo imprinted.

Basic Business is one of a family of accounting and business software packages, and has sold previously for several hundred dollars per module. It has been improved, updated and re-packaged to sell at a market-busting $89.95 for all seven modules. A Point-of-Purchase module, which controls an electronic cash drawer and allows direct entry of transactions from your sales counter is available. Also a dBase file format program is available for importing Basic Business data files into dBase II/III for custom report generation or other special uses.

SOURCE CODE AVAILABLE

Do you have a distinct accounting problem that off-the-shelf software won't handle? Special forms or statements? Don't write your own accounting system from the ground up - start with Basic Business. Call (714) 630-0446 for all the details on source code licensing.

Basic Business can go to work for you today and is available for most popular MS-DOS (IBM and compatibles) and CP/M-80 personal computers. Compare our price, features and attention to detail. There is only one choice. . . it's Basic Business.

Basic Business	$89.95
Point-of-Purchase module	.$99.95
dBase II/III file formats	$19.95

Minimum hardware Required for MS-DOS : 128K memory, two 360K floppy drives (hard disk recommended for Sales and Purchase Order Processing), 132 column printer, MS-DOS (or PC-DOS) version 2.0 or later.

Minimum Hardware Required for CP/M-80 computers: 80 x 24 character display terminal, 64K memory, two 360K disk drives (hard disk recommended for Sales and Purchase Order Processing), 132 column printer.

Circle **no. 176** on reader service card.

SERVICES

File Transfer Service: Advent provides a service beyond the ability of any format conversion software! We can transfer files between MS-DOS/PC-DOS, CP/M and other operating systems in 300 different 3 1/2", 5 1/4" and 8" formats. Includes Apple and Mac, Apricot, Data General One, Kaypro 2000, Eagle, Epson QX-10 & PX-8 (ROM), HP-150, and North Star computers.

ENGINEERING SOFTWARE

ACNAP: A stand-alone Electronics Circuit Analysis Program for use with passive and active circuits consisting of resistors, capacitors, inductors, transistors, op-amps, FETs, etc. Features menu driven and very fast processing times with circuits saved to disk for later use or editing.
ACNAP (CP/M & MS-DOS) **$69.95**

DCNAP: Stand-alone DC circuit analysis program for use with passive and active circuits containing resistors, voltage sources, independent and dependent current sources. Fast, menu- driven program with circuit saved to disk for later use or editing.
DCNAP (CP/M & MS-DOS) **$69.95**

Plotpro: Scientific graph printing program. Prints on 80 or 132 column printer. Create linear, semi-logarithmic, and full logarithmic plots with one or two Y axes in auto or forced scale.
Plotpro (CP/M & MS-DOS) **$69.95**

SPP: This Signal Processing Program contains an integrated set of routines which analyze linear and non-linear systems and circuits and their effects on user specified time domain waveforms. Based on a 512 point Fast Fourier Transform and its inverse. Linear processing is in frequency domain and non-linear processing is in time domain.
SPP (CP/M & MS-DOS) **$69.95**

SOFTWARE UTILITIES

Autodiff: File difference detector. This program finds insertions, deletions, and changes between any two files. Autodiff can mark the file, display, or print the differences, and more!
Autodiff (CP/M) **$29.95**

CP/M DateStamper: Automatically stamp your files with the date it is created, last read, or modified. Works without a Real Time Clock , or with many clocks currently on the market. Utilities are included to allow copying, erasing, or renaming files based on time and date. A time logging utility is included to record computer usage for business/tax purposes.
DateStamper (CP/M) **$49.95**

Media Master +: Read and Write up to 75 CP/M, MS-DOS & TRSDOS disk formats on your IBM or look-alike computer. ZP/EM program is bundled with Media Master to allow CP/M programs to run directly on your MS-DOS computer. An $80.00 value.
Media Master + ... **$59.95**

Pack and Crypt: Two program set. Pack compresses and expands files on disk to save space. Crypt encodes files to provide security for sensitive data. Both are ideal for use with modem transfers.
Pack and Crypt (CP/M & MS-DOS) **$24.95**

Sidekick: One of the most popular programs ever written. Use Sidekick as a calculator, notepad, appointment calendar, auto dialer, ASCII conversion table and much more. On-line help if you forget any of Sidekick's many functions.
Sidekick (MS-DOS) **$54.95**

SmartKey II: New Release! Same great time saver as the original, and allows compiling of definitions you set up with your word-processor! Makes every software program you use easier. Can reduce keystrokes by more than 50% by redefining any key on your keyboard to be any combination of characters or commands that you desire.
SmartKey II **$49.95**

SmartPrint: A powerful add-on to SmartKey, SmartPrint is a vesatile writing tool designed to give you full access to your printer's features such as wide, bold, condensed, underlined, subscript, superscript, and more. Works great with programs like WordStar and others.
SmartPrint **$29.95**

Uniform: Your Computer can read and write up to 80 CP/M, MS-DOS / PC-DOS & TRSDOS disk formats. Versions available for most popular CP/M and MS-DOS computers. Specify your host computer when ordering.
Uniform (CP/M & MS-DOS) **$69.95**

XTREE: Directory maintenance program that graphically displays subdirectories and filename paths. Complete control of your directory including delete, rename, view, list or show. A must for your IBM or compatible.
XTREE (MS-DOS) **$49.95**

Super Zap: Disk patch and dump program. If you have used DU, you will love this menu driven marvel!
Super Zap (CP/M) **$24.95**

ZP/EM: Run almost any CP/M program on your IBM or clone. Use with Media Master or Uniform to allow programs on CP/M disk formats to run directly on your IBM or compatible computer.
ZP/EM (MS-DOS) **$39.95**

FX, QX-10, PX-8 - Epson Corp; CP/M - DRI; MS-DOS - MicroSoft; PC-DOS - IBM Corp.; dBASE II & dBase III - Ashton-Tate; WordStar - MicroPro; UNIX - Bell Laboratories; Apple - Apple Computer Inc.; Basic Business - Advent Products Inc.

PROGRAMMING LANGUAGES

C/80 Ver 3.1: Full featured C compiler and runtime library. One of the fastest on the market. Mathpak is included for true 32 bit floating point and signed integers.
C/80 Ver. 3.1 (CP/M) **$79.90**

C/NIX: Operating System Enhancement for CP/M. C/NIX gives your system many features in the UNIX OS such a hierarchical directory, I/O redirection, "pipes" & "filters" and command files. Uses only 2.3K of TPA and 42K of disk. Requires CP/M 2.x.
C/NIX **$59.95**

LISP/80: Experiment with the artificial intelligence language. Based on the INTERLISP dialect, LISP/80 offers over 75 built-in functions, including file I/O, and string operations. Complete with 36 page manual and demo programs.
LISP/80 .... **$39.95**

Toolworks C: This compiler is a complete subset of C. The two- pass compiler produces relocatable object files (.obj) which are compatible with the MS-DOS LINK program. Mathpak is included for true 32 bit floating point and signed integers.
Toolworks C Compiler (MS-DOS) **$79.90**

Turbo Pascal: Borland version 3.0. The best Pascal compiler on the market.
Turbo Pascal (CP/M & MS-DOS) **$69.95**

Turbo Toolbox: Set of 3 utilities for use with Turbo Pascal.
Turbo Toolbox (CP/M & MS-DOS) **$54.95**

Turbo Tutor: Teaches step-by-step how to use Turbo Pascal.
Turbo Tutor (CP/M & MS-DOS) **$34.95**

Turbo Graphics: Provides full graphics management for producing windows, pie and pie charts, circles and other geometric shapes with Turbo Pascal.
Turbo Graphics (MS-DOS) **$54.95**

TEXT EDITING

Punctuation & Style: Improves your writing by catching unbalanced quotes, parentheses and brackets, improper abbreviations, capitalization, sentence structure, much more. It's like having your own copy editor!
Punctuation & Style (CP/M & MS-DOS) **$125.00**

Word Finder: This powerful 90,000 word Thesaurus allows you to select the best word for the application. Works inside WordStar for greater ease of use. Instantly searches its dictionary, then displays synonyms, and automatically deletes the "wrong" word and replaces it with the "right" word. Requires 380K disk storage.
Word Finder (CP/M) **$79.95**

Wordpatch: Print files with tiny, compressed, wide, or wide compressed type faces, 5 sizes of italic, real superscripts and subscripts, and 6, 7, and 8 lines per inch spacing. No new print controls to learn. Supports most popular dot matrix printers. A must for WordStar users!
Wordpatch (CP/M & MS-DOS) **$49.95**

The Word Plus: The ultimate spelling checker. Not only finds misspelled words but shows you correct spelling options, shows the word in context, allows you to build dictionaries of special words you use, and much more.
The Word Plus (MS-DOS) **$150.00**

HARDWARE & SUPPLIES

Finger Print "Letter Writer": Unleash your Epson FX series printer. Add near-letter-quality print, IBM and/or Apple Graphics printer emulation, plus 16 other print functions! Three replacement chips quickly fit inside Epson FX series printers. Easy installation. Does not void printer warranty.
Finger Print "LetterWriter" **$79.95**

Diskettes, Double Density:

Maxell 10-pack w/ storage box:		3M box of 10:	
Single Sided	**$19.95**	**Single Sided**	**$22.95**
Double Sided	**$23.95**	**Double Sided**	**$26.95**

Economy Diskettes: package of 25 including tyvek sleeves.
Single Sided **$29.50**
Double Sided **$31.25**

Call or write for our FREE catalog

All items are warranteed for 90 days. 30 day money back guarantee if not completely satisfied. Guarantee for software applies only if diskette seal is intact. Visa and MasterCard are welcome. Please add 2.00 freight per total order and 2.00 for COD orders. California residents please add 6% sales tax. Prices, availability and specifications subject to change without notice.

CALL TODAY **National** **(800) 821-8778**
California **(800) 521-7182**

Hours: Mon - Fri 8 am - 5 pm PDT

DEALER INQUIRIES WELCOME.

advent products inc.

3154-F E. La Palma Ave
Anaheim, CA 92806
(714) 630-0446

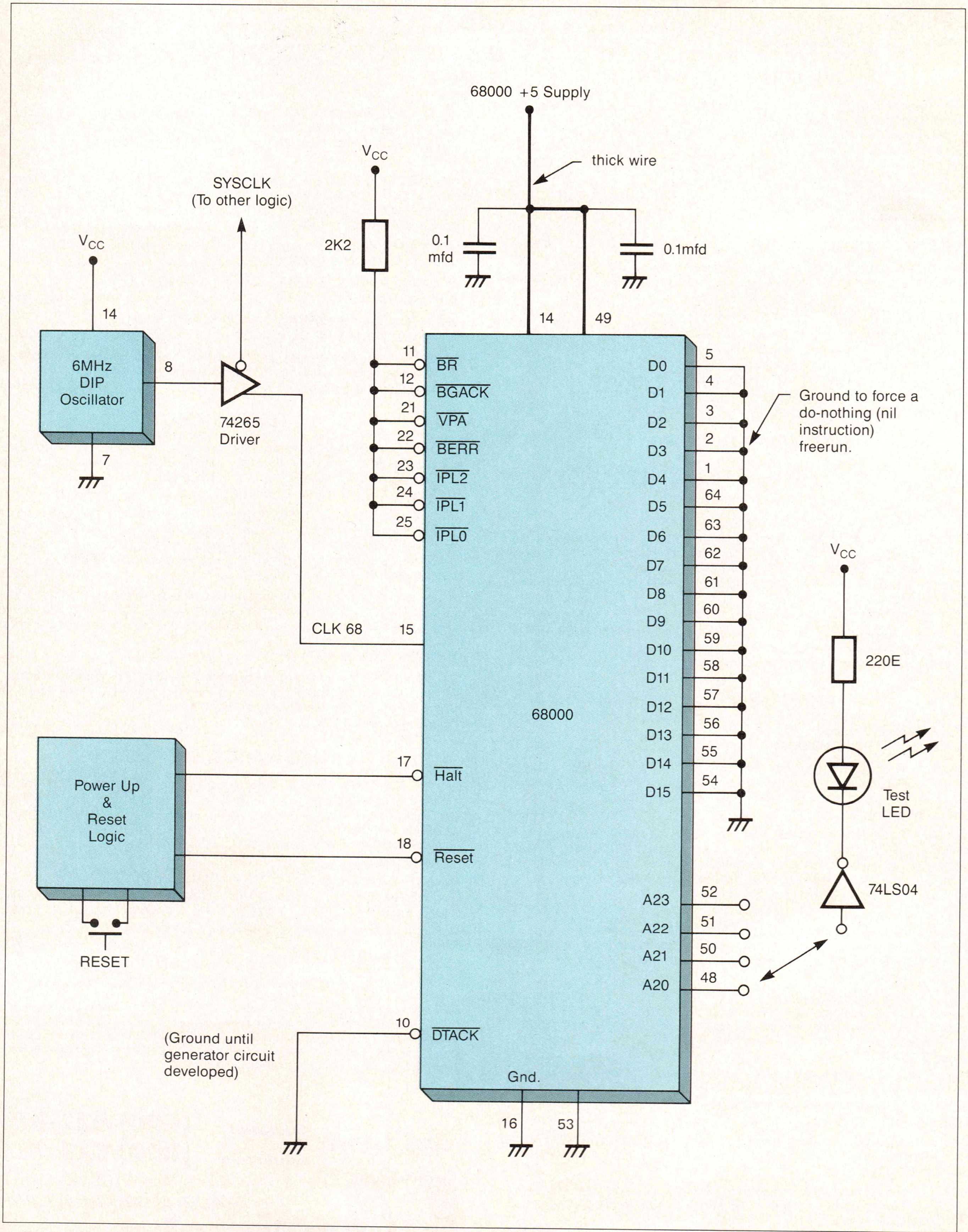


Figure 4: *Circuit diagram of the minimum 68000 system for free-run test*

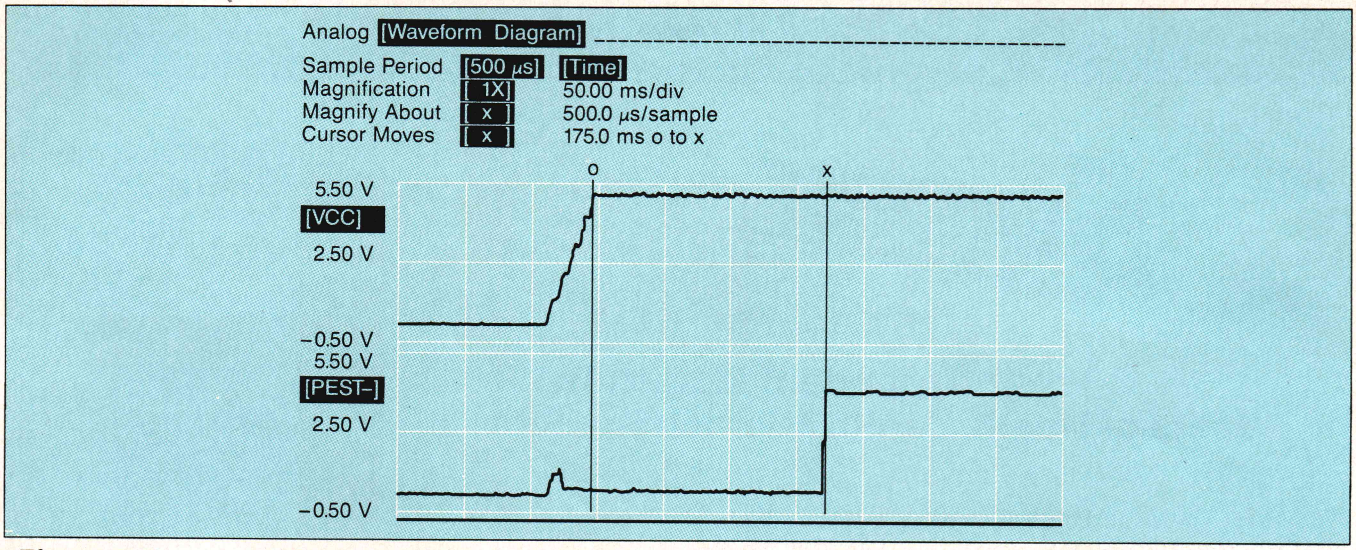


Figure 5: *Power-up performance of the 555 timer circuit. On power-up, the 555 timer with the parts given in the schematic provides about 175 ms RESET* to the 68000.*

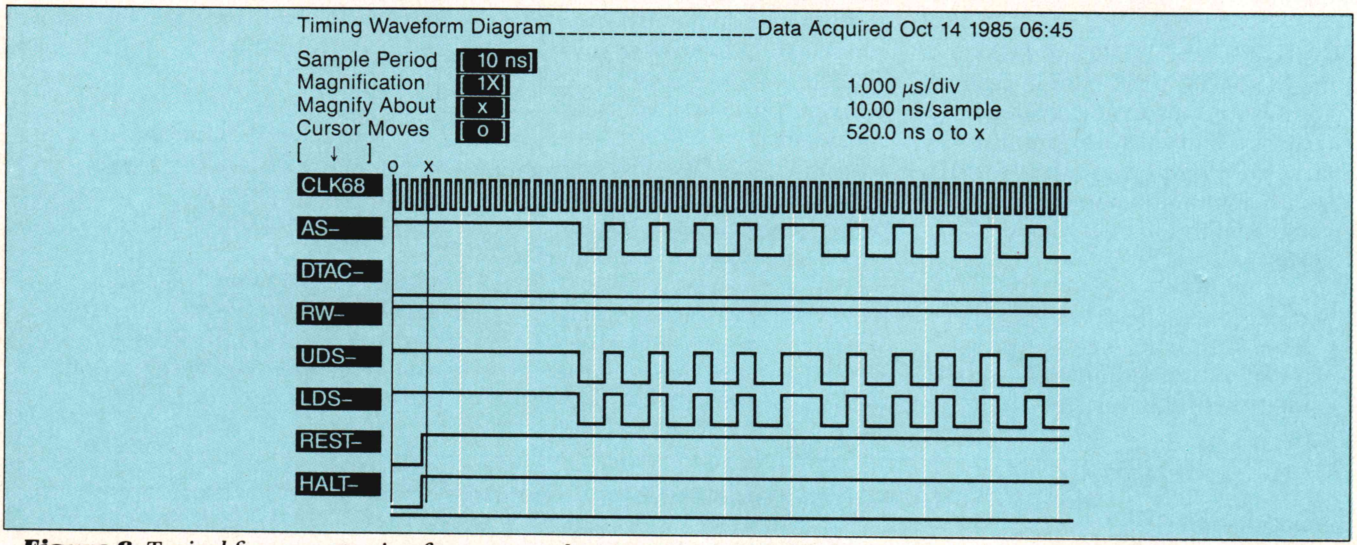


Figure 6: *Typical free run starting from a complete RESET* and HALT* asserted low. The clock is running at 6 MHz. DTACK* is grounded in this example.*

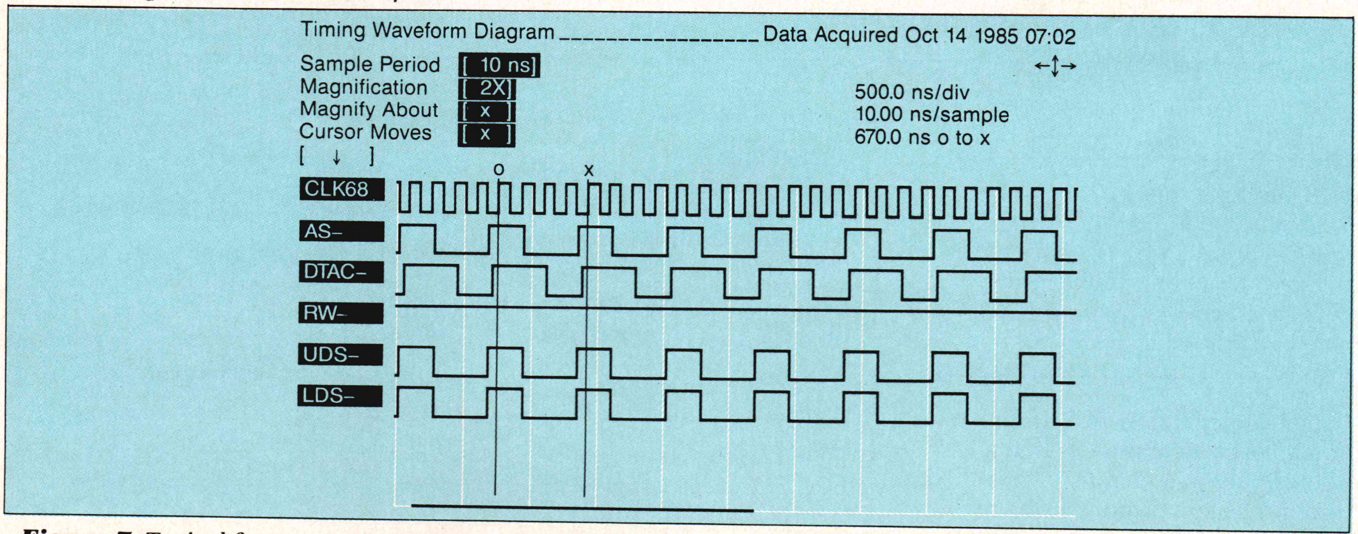
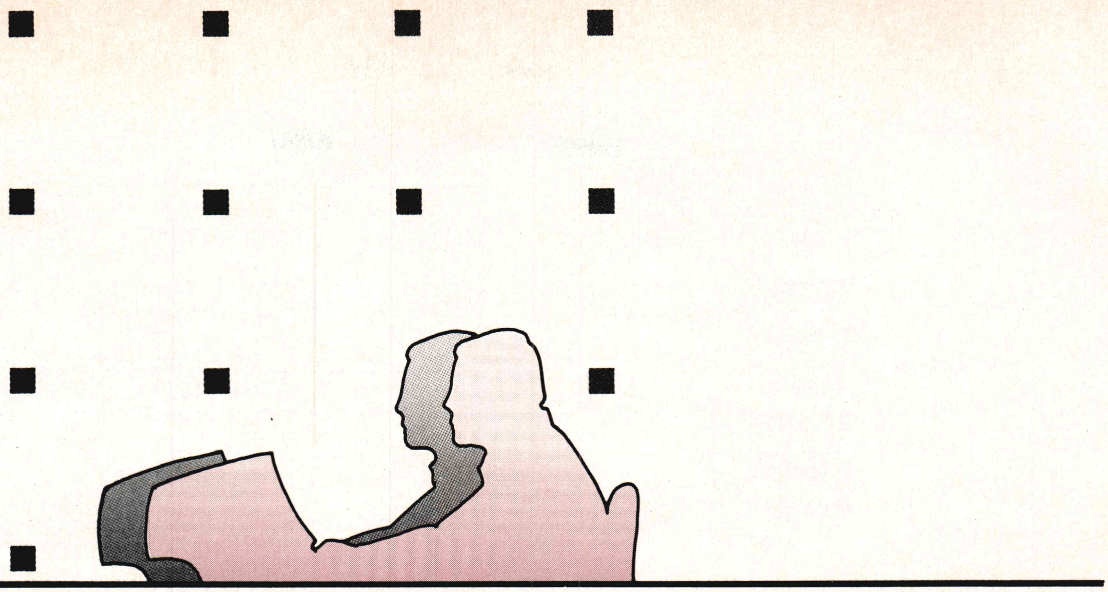


Figure 7: *Typical free run with the DTACK* circuit enabled. The clock is 6 MHz, and there are no wait states inserted.*



THE International UNIX* Event of the Year!

February 4-7, 1986 / Anaheim Convention Center / Anaheim, California

UniForum 1986, the International Conference of UNIX Users, has established itself as THE premier UNIX conference/trade show.

■ Over 200 major vendors exhibiting their newest UNIX-based hardware, software, systems, services and peripherals.

■ A complete tutorial and conference program. Multiple sessions target the connectivity of UNIX between the user and technical environments; the hardware and software technology for office systems and workstations; the interface between man and machine, machine and machine, and much more. Day-long tutorials provide intensive, focused material on specific subjects in UNIX...while the conference sessions highlight the latest developments in the continuing evolution of UNIX.

*UNIX is a trademark of AT&T Bell Laboratories.

■ **FREE** UNIX Introductory Workshops.

■ Birds-of-a-Feather impromptu sessions.

Call us NOW for your **FREE** Show-Only Badge *and* complete registration information on the dynamic conference/tutorial program.

The power and potential of UNIX are important to you. UniForum 1986 is THE computer event you can't afford to miss!

UniForum™
The International Conference of UNIX Users

For Complete Information, Call:

800-323-5155

(In Illinois, Call: 312-299-3131)

Or Write:

UniForum
2400 East Devon Avenue
Suite 205
Des Plaines, IL 60018

Sponsored by

/usr/group
The International Network of UNIX Users

Circle **no. 120** on reader service card.

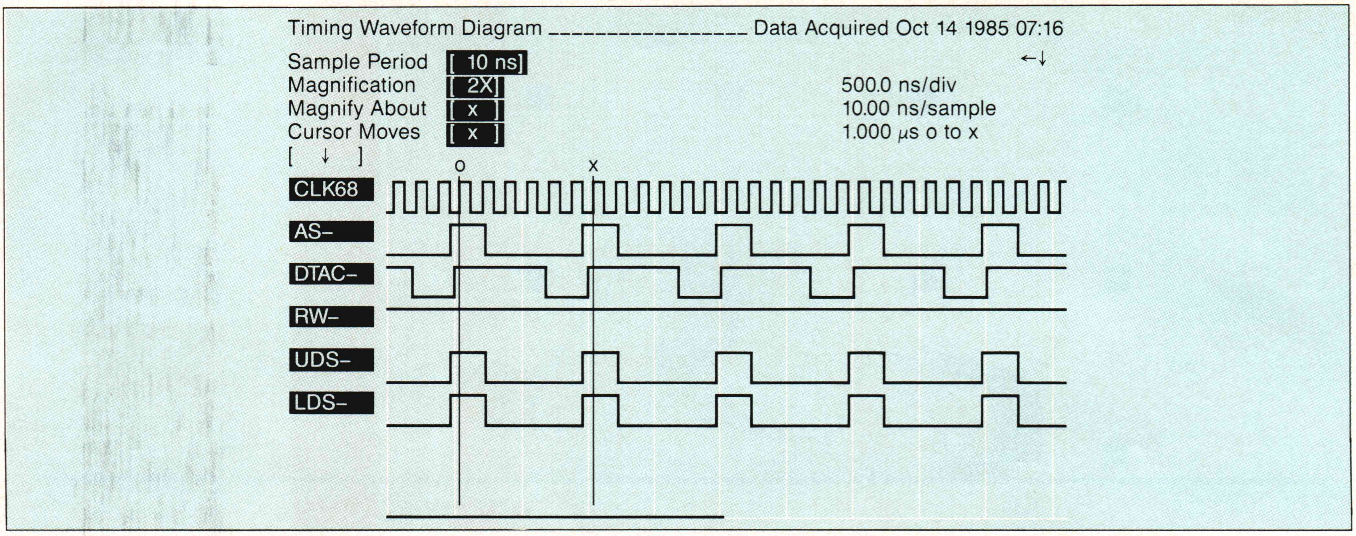


Figure 8: *Typical free run with DTACK* enabled. This timing shows DTACK* delayed enough to cause two waits in each bus cycle.*

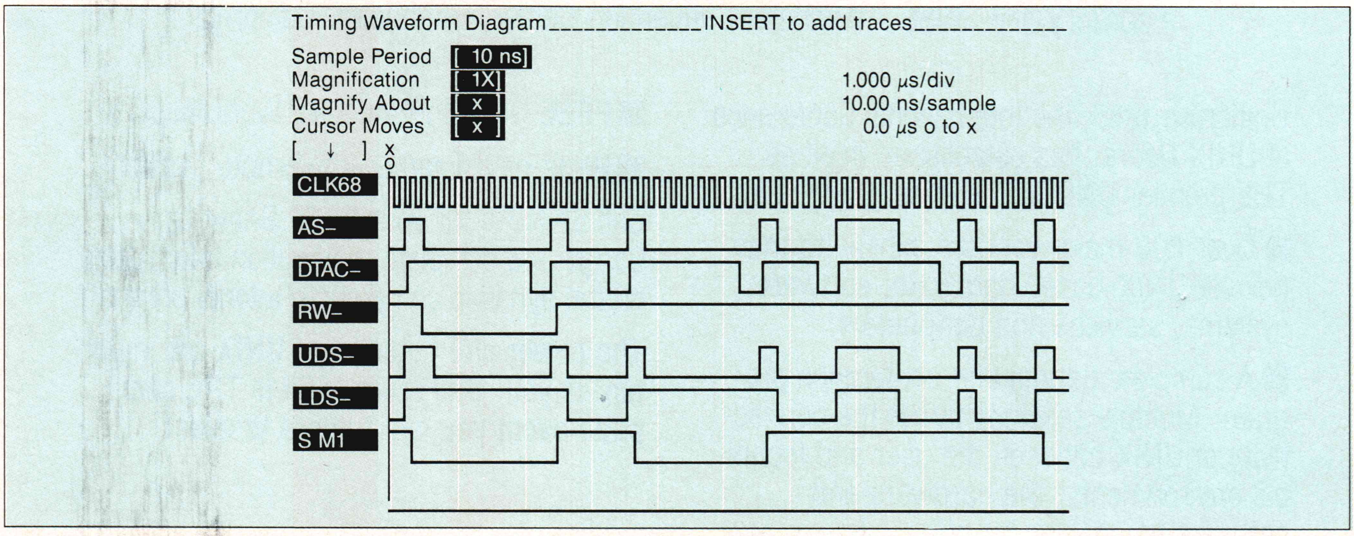


Figure 9: *A view of the bus activity when the TUTOR EPROM set runs at 6 MHz. The CPU board was set for eight waits on I/O and three waits otherwise.*

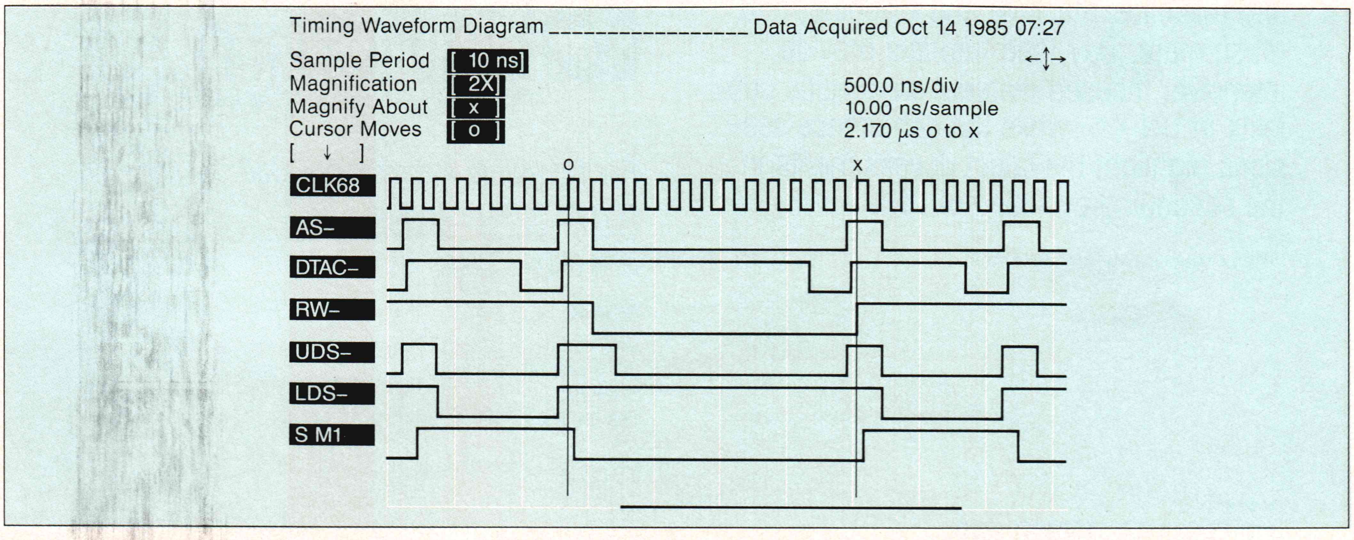


Figure 10: *A closer look at the bus controls when TUTOR executes a write bus cycle*

Now You Know Why BRIEF™ is BEST

"BRIEF, The Programmer's Editor, is simply the best text editor you can buy." John Dvorak, INFOWORLD 7/8/85

The Program Editor with the BEST Features

Since its introduction, BRIEF has been sweeping programmers off their feet. Why? Because BRIEF offers the features **MOST ASKED FOR** by professional programmers. In fact, BRIEF has just about every feature you've ever seen or imagined, including the ability to configure windows, keyboard assignments, and commands to **YOUR** preference. One reviewer (David Irwin, DATA BASED ADVISOR) put it most aptly, "(BRIEF)...is quite simply the best code editor I have seen."

"A bona fide Undo..."

Steve McMahon, BYTE 3/85

As Mark Edwards describes in DR. DOBB'S JOURNAL (11/85), "BRIEF has an outstanding undo facility. The default configuration allows the last 30 editing commands to be undone. This number can be raised to a maximum of 300 commands. Until you reach this maximum or run out of RAM, every command you issue can be undone. So if you make ten changes and then realize that the first one was an error, you can undo all the changes back to the mistake... Needless to emphasize, this facility can save endless grief."

No other editor has this capability.

Every Feature You Can Imagine

Compare these features with your editor (or any other for that matter).

- FAST
- Full UNDO (N Times)
- Edit Multiple Large Files
- Compiler-specific support like auto indent, syntax check, compile within BRIEF, and template editing
- Exit to DOS inside BRIEF
- Uses all Available Memory
- Tutorial
- Repeat Keystroke Sequences
- 15 Minute Learning Time
- Windows (Tiled and Pop-up)
- Unlimited File Size –(even 2 Meg!)
- Reconfigurable Keyboard
- Context Sensitive Help
- Search for "regular expressions"
- Mnemonic Key Assignments
- Horizontal Scrolling
- Comprehensive Error Recovery
- A Complete Compiled Programmable and Readable Macro Language
- EGA and Large Display Support
- Adjustable line length up to 512

Program Editing YOUR Way

A typical program editor requires you to adjust your style of programming to its particular requirements – NOT SO WITH BRIEF. You can easily customize BRIEF to your way of doing things, making it a natural extension of your mind. For example, you can create ANY command and assign it to ANY key – even basic function keys such as cursor-control keys or the return key.

The Experts Agree

Reviewers at BYTE, INFOWORLD, DATA BASED ADVISOR, and DR. DOBB'S JOURNAL all came to the same conclusion – BRIEF IS BEST!

Further, of 20 top industry experts who were given BRIEF to test, 15 were so impressed they scrapped their existing editors!

NOT COPY PROTECTED

Solution Systems™

MONEY-BACK GUARANTEE
Try BRIEF ($195) for 30 days – If not satisfied get a full refund.
TO ORDER CALL (800-821-2492)

SOLUTION SYSTEMS, 335-D WASHINGTON ST., NORWELL, MA 02061, 617-659-1571

BRIEF is a trademark of UnderWare

68000
(Continued from page 62)

tween the two clocks of some 10 to 15 ns, depending on loading. Although this amount of skew seems slight, it can cause severe timing difficulties when the clock speed gets above 10 MHz. The 74265 quad complementary-output element with a worst-case skew of 3 ns is a good selection; in my 12-MHz prototype, this selection has worked out well.

The reset and halt module has two basic functions. One task is to hold the 68000 HALT* and RESET* lines low for at least 100 ms on power-up. Its other function is to pull the same two lines low for at least ten clock cycles for a reset button press at any time after the power has been on.

The circuit in Figure 3 (page 62) provides a simple and reliable reset function for the 68000. It provides a reset pulse either on power-up or whenever you press the reset button. Open-collector devices are required because the 68000 HALT* and RESET* controls are both bidirectional. For example, the 68000 can itself drive the RESET* line to reset any peripherals if the software reset instruction is issued. Also, the 68000 can force the HALT* line low if the system cannot continue processing. A single "halt" LED connected as shown is valuable in helping bring up the processor for the first time.

The last module in the minimum system is the 68000 processor shown in Figure 4 (page 66). By now, you should have checked the power, clock, and reset modules for proper operation and connected them ready for the 68000. If the processor is wired as shown, it should begin free running immediately. On power-up the HALT light should flash briefly, and then the TEST light will begin flashing on and off.

Earlier I referred to my so-called NIL instruction. As you see in the circuit, the data bus is completely grounded so the NIL has an opcode of 0000. In the context of its use in free running, it acts like a no-operation or NOP. The 68000 does have a NOP opcode ($4E71), but this NOP will not work as a free-running instruction.

A critical constraint on the opcode precludes using the NOP instruction in free running: Whatever is wired

to the data bus for the 68000 to read upon reset must be even. The reset sequence is this: The 68000 will do four 16-bit reads to get the initial SSP and PC vectors; then it will fetch its first opcode at the address in the PC. If the PC is not aligned on an even address, the 68000 detects an address error and immediately begins illegal-address exception processing. It tries to push its status on the stack at the beginning of the exception, but the stack is also an illegal address (the same noneven number as in the PC). The result is the fatal double bus fault that stops all processing and asserts the HALT* output.

The opcode 0000 does in fact correspond to a real instruction in the 68000 set. It is the mnemonic ORI.B #0,D0, and it was selected for free running for two reasons: first, because it was even; and second, because connecting all grounds to the data bus was simpler than making sure one or two data lines had a logic 1 on them. When the instruction is considered in its free-running environment, the appearance of its memory is as shown in Table 1 (page 61).

You can calculate the execution time of this "program" easily. Each instruction takes eight clock cycles (two read bus cycles), so for a 6-MHz clock, the execution time is 8×167 ns or approximately 1.33 microseconds. A complete sweep through the entire 16 megabytes of the 68000 address range takes 1.33×4 megabytes or about 5.59 seconds. If you connect the TEST light to the top address bit, A23, it will be on for 2.8 seconds and then off for 2.8 seconds. I connected the TEST light to A20 permanently. It stays on for 0.35 seconds and off for 0.35 seconds—a reassuring flash rate during development work and not nearly as unsettling as a constant red HALT light.

Results

Figure 5 (page 68) shows the performance of the power-up timer circuit. The top plot is the main system power as it comes on and eventually regulates at 5 volts on the CPU board. About 175 ms after the supply voltage is valid, the RESET* and HALT* lines go high to successfully complete a full 68000 reset.

The effect of this reset operation is shown in Figure 6 (page 68). The last two lines on the timing diagram are the RESET* and HALT* controls for the 68000. After its internal start-up time, the 68000 asserts its address strobe (AS*) and its data strobe lines (UDS* and LDS*) in the first read bus cycle. After four read bus cycles, the processor PC begins execution at address 0, as discussed above.

DTACK* is the asynchronous bus control line that normally comes back from memory or peripherals to tell the 68000 to complete the current bus cycle. During the initial free run of the processor, there is nothing connected that will acknowledge a data transfer, so the control is grounded. The timing diagram shows this line at a logic low. The timing diagram also shows the read/

68000

(Continued from page 73)

write control, R/W*, as constantly high because the 68000 only does a read bus cycle when it is free running.

Figure 7 (page 68) shows the free-running processor with a DTACK* generator in operation. Notice the o and x markers bracketing a single bus cycle. The normal read bus cycle has a total of four clock cycles. If DTACK* is delayed for two cycles, as shown in Figure 8 (page 70), then the bus cycle is lengthened and two "waits" are inserted into each bus cycle. When you interface memory or peripherals to the 68000, you can design each external module to hold back DTACK* until its unique timing requirements are met.

As an example, the timing diagram in Figure 9 (page 70) shows the system while *not* free running: It is executing the monitor program (TUTOR) and waiting for a keystroke. The system was set to provide eight waits for I/O read operations, nine for writes, and three waits otherwise. Figure 10 (page 70) shows a close-up look at the bus cycles. The lower timing line, marked sM1, is the S-100 bus status indicating an opcode fetch.

Summary

Bringing up the 68000 using the free-running technique is very different from the more traditional approaches to getting a processor running. You can see, though, just by the brief description of this first step in bringing up the 68000 kernel, that you do not need sophisticated equipment to get started.

There is more to be said about all the steps beyond this first free-running processor; no doubt many questions remain unanswered. From my experience, though, the understanding you can get from doing a free-running 68000 is very valuable, and it can help you go on to design and build a complete system successfully.

Availability

The TUTOR firmware is available directly from the author.

DDJ

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 4.**

COM: An 8080 Simulator for the MC68000

by Jim Cathey

Simulation time can vary widely, as some 8080 instructions aren't easily simulated with the 68000.

After I bought a 68000-based S-100 system, I found that I needed to run several CP/M 2.2 (alias CP/M-80) software packages, none of which were available in equivalent forms for my machine. I was faced with two options—either buying another processor board so that I could run these programs or writing an 8080 simulator. The software approach seemed infinitely preferable. This also seemed like an opportune time to find out just how fast my 68000 really was. So I set out to write an 8080 simulator (which I named COM because it interprets .com files).

In principle, writing a simulator is simple. You just set up a set of fake registers and start picking up opcodes and interpreting them. Unfortunately, it's the little details that get you. This simulator took about twice as long to write as I expected. As it is, it isn't perfect. In fact, it would slow down considerably if it were. However, most programs aren't bothered by the imperfections, and the speed difference would be significant because the simulation is already on the slow side of usable. I wrote COM to run as fast as possible.

The simulation speed is approximately that of a 1.4-MHz Z80 processor based on a sample assembly with MAC. (My 68000 system is an 8-MHz CompuPro/Morrow single-user hybrid running CP/M-68K 1.2, with 16-bit no-wait-state memory.) Simulation time can vary widely, as some 8080 instructions aren't easily simulated with the 68000. MAC was chosen as a typical example program. LU, the public-domain library utility, is one of the worst performers. It spends vast amounts of time calculating CRCs. This instruction sequence isn't very efficient on the 8080 (using a C arithmetic library), and the simulation magnifies any inefficiencies. I didn't time LU for comparison, but I thought that the simulation had crashed because nothing seemed to be happening for long periods. Another poor performer is WordStar. I tried simulating it because it is so popular and because it gives the simulator a thorough workout. The simulation works, and it does so at an acceptable performance level most of the time.

About the Program

The program source is broken into four files. Listing One (page 104) is the first file, which contains the start-up, command-line, CP/M-2.2 simulation, and trace routines. Listing Two (page 112) contains opcode simulation subroutines and flag tables. The rest of the listings will be continued in the March issue.

The program starts out by prompting for the trace end points if that code has been included. (There are several conditional assembly trace features in COM.) It then builds the 8080 environment in a 64K buffer (biggest TPA yet!) and initializes the 68000 simulation registers, with the 8080's PC set to 100H into the buffer. It then calls a subroutine that loads the specified .com program into the buffer and transfers the 68000's second FCB to the 8080's first FCB and passes the remains of the command tail to the 8080's DMA buffer. If all goes well, the program then enters the main loop at label MLOOP. Here it fetches the next 8080 opcode and uses it to index into a table of 256 68000 subroutines (one per opcode—big, but fast) and jumps to the selected subroutine. Each opcode subroutine then picks up what parameters it needs and plays with the fake registers appropriately. Each subroutine then jumps back to MLOOP, which repeats the process. This continues until a service request is picked up or until an illegal instruction is found.

The service request used by the simulation is the HLT opcode ($76). HLT is followed by a 1-byte parameter telling the 68000 which action to take. All BIOS/BDOS functions are implemented as service requests. After the service is performed, execution continues at MLOOP (or at the byte following the parameter—it depends on your point of view). Refer to the 8080 Environment section for more details on how service requests are used by the 8080's BDOS/BIOS.

Register dumps are caused by illegal opcodes or are done during a trace. They are easily interpreted (registers S0-S3 are the top four en-

Jim Cathey, ISC Systems Corp., TAF-C8, Spokane, WA 99220.

tries on the stack) as the current instruction is also disassembled. Illegal opcodes terminate the simulation after the register dump.

Flag simulation is done with two tables. Because any 8080 logical operation that sets the parity flag also clears the carry bit, these flag results are based solely on the value in the accumulator. The simulator uses a 256-byte flag lookup table for these operations. Similarly, anything that conditionally sets carry (an arithmetic operation) doesn't need to set the parity flag if the code is intended to run on a Z80. (This describes all CP/M-2.2 software I was interested in.) Another 16-byte table can therefore be used for arithmetic flag results. The 4-bit flag field of the 68000's status register is used as the index for this second table. The 68000 does have an overflow flag, so this is substituted for the parity bit of the 8080 (exactly as in the Z80). This causes one problem that is discussed in the Known Faults section. Treatment of the half carry bit is also discussed later because it doesn't fit into either of the tables.

The CP/M-80 environment simulation is greatly simplified (to my great disappointment) by the strong resemblance of CP/M-68K to CP/M-80. Most of the calls are directly translatable. There are a few exceptions, though, and they require the bulk of the code.

1. Any call referencing an FCB requires that the byte order of the Random Record field be switched, if the call uses that field.

2. CP/M-68K can't open a file in any but the base extent. You have to change such requests to an open in the base extent and then do a Random Read to the point you wanted.

3. Direct console I/O (BDOS #6) under CP/M-80 returns a null flag if no character is available. CP/M-68K waits for a character. A status check is performed first, and if a character is ready, it is fetched and returned to the simulation. Otherwise just a null status is returned.

4. Any call referencing an address (DMA or otherwise) needs to have that address translated to point into the 8080's code buffer. (This is a problem inherent to the simulation because the 8080 buffer cannot be placed in system memory at address 0.)

Some instruction simulations don't do what you think they might. Specifically, the EI and DI instructions do not translate to the equivalent 68000 sequences. The object of using DI/EI pairs in an 8080 program is to prevent time-critical code from being interrupted or to prevent resource contention between interrupt routines and background processes that share resources. Because there are no 8080 "interrupts" possible under COM, there is nothing to block. (We won't even talk about simulating time-critical code!)

A few Z80 simulation routines are used in COM because of the overflow/parity flag problem discussed below. These are just an extension of the table approach used for the 8080—a large jump table and a bunch of subroutines. Extension to a full Z80 simulation is straightforward but would require a lot of code if the present jump table technique is kept.

Another approach to simulating instructions involves dividing the opcodes into classes, e.g., all MOVs handled by one subroutine that figures

(Continued from page 77)

out what to move where by examining the opcode in further detail. Though this is much smaller codewise than having the 63 similar subroutines that I used, it also suffers a speed penalty that I just couldn't tolerate. I didn't buy a 68000 just to slow it down!

Known Faults

There are two problems with the simulation of the 8080's flags. The first is that the flags are more like those of the Z80 than the 8080 in that the parity flag reflects overflow status after arithmetic operations rather than parity. This fools some dynamically selected run-time packages such as the one used for BDS-C. There is minimal Z80 support in COM to handle the few extra instructions that BDS-C wants to use (LDIR and LDDR) so that programs compiled by it will run. (CPIR was also required by another program for the same reason.) This Z80 support could be extended as much as needed—up to and including full Z80 simulation.

The other flag problem is with the half carry bit. Simulating it would take a lot of overhead because there is no similar flag in the 68000. Therefore, assuming the only need for the flag is for the DAA instruction, this instruction is treated specially. Instructions that set the half carry bit meaningfully (ADDs, ADCs, and INR As) have additional code to store away the two operands and CY (if used) in special locations. The DAA simulation then recreates the HCY bit out of these stored values. A problem can arise when the flags are pushed and then pulled before the DAA is executed—an incorrect HCY is created if another addition-type operation occurs while the flags are supposedly saved. In practice, I only ran into this problem when using the 8080 DDT to trace through a DAA instruction. Be forewarned. If required, the simulation could be extended to eliminate this problem by proper simulation of the HCY bit, but this would slow the arithmetic routines down quite a bit.

BIOS calls to the disk drivers aren't allowed. I did this for safety reasons more than anything else—I didn't want possibly buggy simulations playing with my hard disk without the protection of my BDOS. This limitation could easily be removed.

BDOS call #31 (Get DPH address) isn't supported. I didn't need this for any program I wanted to use so it was left out. Including it would involve getting the table from the 68000, copying it to somewhere where the "8080" could find it, and then returning a pointer to this copy of the table. Similarly, function #27 (Get ALLOC vector) isn't supported either. Using either of these calls will cause an abort and an appropriate error message.

The IOBYTE and LOGIN vectors at 3 and 4 aren't supported.

Only one parsed FCB is supplied in the 8080's base page. The normally present second name at $6C isn't parsed. (Note that CP/M-68K parses two full FCBs when COM is invoked. The first is the name of the .com program to run, and the second is used as this program's first FCB. CP/M-80's second "FCB" isn't one—it is only an-

```
            ORG 0
WARMST:     JMP BIOS

            ORG 5
            JMP BDOS

            ORG 0FF00H
BDOS:       HLT              ; Service request.
            DB 0             ; 0 is BDOS call, else BIOS.
            RET

BIOS:       JMP WBOOT
            JMP CONST
            JMP CONIN
            JMP CONOUT
            JMP LIST
            JMP PUNCH
            JMP READER
            JMP HOME
            JMP SELDSK
            JMP SETTRK
            JMP SETSEC
            JMP SETDMA
            JMP READ
            JMP WRITE
            JMP LISTST
            JMP SECTRN

WBOOT:      HLT              ; Service Request
            DB 1             ; Passed to 68K BIOS (BIOS #1)
            RET

CONST:      HLT
            DB 2             ; BIOS #2 (etc.)
            RET

CONIN:      HLT
            DB 3
            RET

CONOUT:     HLT
            DB 4
            RET

LIST:       HLT
            DB 5
            RET
```

Table 1

other name field in the first FCB. I didn't find anything that needed it, so I didn't go to the trouble of picking another name out of the 68K's command tail.)

An additional complication inherent to the simulation arises when you try to use programs running under COM to drive DMA devices. The hard disk controller in my system is a Morrow HD/DMA. In order for the 8080 simulation to be able to drive this controller, all addresses passed to the board's DMA circuitry must undergo translation so that they point to the buffers in the "8080" program space, not the 68000's! Handling this tends to require a lot of code specific to each device supported, but it can be done. (COM was originally written for two reasons: to develop firmware for the 8085 used in my system's keyboard and to run the Morrow FORMATMW [hard disk formatter] program. Tracing this program pointed out the error in the HD/DMA documentation that kept my own formatting program from working.) There is a conditional assembly flag in COM to include the support for the Morrow controller. You probably will never want this option, but I left the code in to serve as an example of extending COM to support a DMA device.

(Continued from page 78)

```
PUNCH:   HLT
         DB 6
         RET

READER:  HLT
         DB 7
         RET

HOME:    HLT              ; Normally blocked by
         DB 8             ; the simulation.
         RET

SELDSK:  HLT              ; Ditto, etc.
         DB 9
         RET

SETTRK:  HLT
         DB 10
         RET

SETSEC:  HLT
         DB 11
         RET

SETDMA:  HLT
         DB 12
         RET

READ:    HLT
         DB 13
         RET

WRITE:   HLT
         DB 14
         RET

LISTST:  HLT              ; This one is allowed.
         DB 15
         RET

SECTRN:  HLT
         DB 16
         RET
         END
```

Table 1

8080 Environment

The 8080 Environment is a 64K buffer, of which all but 512 bytes are available as TPA. This is probably the only real advantage of the simulation over real execution. The fake BIOS/BDOS (FDOS) starts at 8080 address $FF00 and is in the form of a jump table followed by a service request table. The warmstart and BDOS jumps in the low page of the buffer point to these tables. There is no CCP because COM takes its place. The 8080 form of the FDOS is shown in Table 1 (page 78).

The service request handler performs a BIOS call to the 68000 if the parameter following HLT is not zero or a BDOS call if the parm is zero. In either case the appropriate parameters from the fake 8080 registers are translated (if required) and passed to the 68000 FDOS. The return values are then translated (if required) and stuffed into the 8080's pseudoregisters, and the simulation is continued.

Using COM

CP/M-80 programs are run by inserting the word COM in the normal command line. Examples of use are:

```
A>COM WS TEST.ASM
A>COM MAC TEST
A>COM LOAD TEST
A>COM DDT TEST.COM
A>COM LU -O JUNK -A TEST.COM
A>COM LDIR JUNK
A>COM MBASIC FFT.BAS
```

COM may be assembled with several optional trace facilities. Normally I create a separate version called COMT because the presence of the trace code slows down the simulation. A trace is specified by giving the full normal command line and then answering the prompts for the start and stop trace addresses. COMT will check each address before it simulates the opcode at that address for a match with either of the two limits and turn on or off the register dump appropriately. An example is shown

8080 SIMULATOR
(Continued from page 79)

in Table 2 (page 81). The tracing addresses in this figure will trace every BDOS call made by the program.

The code for address prompting is rather stupid—it doesn't use line-buffered I/O. Because I almost never use tracing now that COM works, I didn't bother to fix this up. (The code was pulled out of my ROM monitor, where I didn't necessarily have any RAM for a buffer, which explains its strange structure.)

Other tracing code may be included in COMT. There is support for dumping (to the printer) FCB calls to the BDOS and for including a register dump at this time. All of these trace options were useful in debugging the simulation. You probably won't need them, but they illustrate one advantage of the simulation over the real thing—you can monitor events at any level of detail you want, provided you don't need real-time execution.

Teaching the Assembler Tricks (With a Hammer)

This section describes some of the tricks I used to make the ALCYON assembler (AS68) that is distributed with the CP/M-68K package do what I wanted when writing COM. Also described are some other tricks that I have used successfully in other assembly programs. (AS68 is also the assembler distributed with the Atari 520ST developer's package.)

At the beginning of Listing One are the register definitions used by COM. The form of these definitions was picked out of the AS68INIT file that you use the first time you run AS68. These definitions allow you to refer to the 68000 registers with more meaningful names than just "D3" and so on. The only real disadvantage of using names is that it is easy to forget which registers are in use and accidentally use one as a temporary that should have been saved first. Proper documentation helps in this. You also cannot use these new names in a "reg" directive.

One problem with these names (and all other "equ" defined symbols) is that you can't define them as global and use them in another file. The declarations must be repeated in each source file.

Free! Product Information

Dr. Dobb's Journal of Software Tools

January 1986 #111 **Expiration Date: April 30, 1986**

Name ______________________ Title ______________________

Company ______________________ Phone ______________________

Address ____

City/State/Zip ____

Please circle one letter in each category:

I. My work is performed:
- A. for in-house use only.
- B. for other companies.
- C. for end users/retailers.
- D. in none of the above areas.

II. My primary job function:
- A. Software Project Mgmt/Spvr
- B. Hardware Project Mgmt/Spvr
- C. Computer Consultant
- D. Corporate Management
- E. Other

III. My company department performs:
- A. software development.
- B. computer system integration.
- C. computer manufacturing.
- D. computer consulting.
- E. computer research.
- F. none of the above.

IV. This inquiry is for:
- A. a purchase within 1 month.
- B. a purchase within 1 to 6 months.
- C. product information only.

V. Corporate Purchase Authority:
- A. Final Decision-maker
- B. Approve/Recommend
- C. No Influence

VI. Personal Computer Users at my Jobsite:
- A. 10,000 or more
- B. 500 to 9,999
- C. 100 to 499
- D. 10 to 99
- E. less than 10

VII. On average, I advise others about computers:
- A. more than once per day.
- B. once per day.
- C. once per week.
- D. less than once per week.

VIII. In my job function, I:
- A. design software and/or write code.
- B. design software.
- C. write code.
- D. don't design software or write code.

A Reader Service number appears on each advertisement. Circle the corresponding numbers below for more info.

001	002	003	004	005	006	007	008	00
010	011	012	013	014	015	016	017	01
019	020	021	022	023	024	025	026	02
028	029	030	031	032	033	034	035	03
037	038	039	040	041	042	043	044	04
046	047	048	049	050	051	052	053	05
055	056	057	058	059	060	061	062	06
064	065	066	067	068	069	070	071	07
073	074	075	076	077	078	079	080	08
082	083	084	085	086	087	088	089	09
091	092	093	094	095	096	097	098	09
100	101	102	103	104	105	106	107	10
109	110	111	112	113	114	115	116	1
118	119	120	121	122	123	124	125	12
127	128	129	130	131	132	133	134	13
136	137	138	139	140	141	142	143	14
145	146	147	148	149	150	151	152	15
154	155	156	157	158	159	160	161	16
163	164	165	166	167	168	169	170	17
172	173	174	175	176	177	178	179	18
181	182	183	184	185	186	187	188	18
190	191	192	193	194	195	196	197	19
199	200	201	202	203	204	205	206	20
208	209	210	211	212	213	214	215	2
217	218	219	220	221	222	223	224	2
226	227	228	229	230	231	232	233	2
235	236	237	238	239	240	241	242	2
244	245	246	247	248	249	250	251	2
253	254	255	256	257	258	259	260	2
262	263	264	265	266	267	268	269	2
271	272	273	274	275	276	277	278	2
280	281	282	283	284	285	286	287	2
289	290	291	292	293	294	295	296	2
298	299	999						

Circle 999 to start a 12 month subscription at the price of $29.97

Thank You!
Dr. Dobb's greatly appreciates your responses to questions I through VIII.

Postage Paid!

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
First Class Permit #217, Clinton, Iowa

POSTAGE WILL BE PAID BY ADDRESSEE

Dr. Dobb's Journal of
Software Tools
P.O. Box 2157
Clinton, Iowa 52735-2157

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES

BUSINESS REPLY MAIL
First Class Permit #217, Clinton, Iowa

POSTAGE WILL BE PAID BY ADDRESSEE

Dr. Dobb's Journal of
Software Tools
P.O. Box 2157
Clinton, Iowa 52735-2157

Thank You!
Dr. Dobb's greatly appreciates your responses to questions I through VIII.

Postage Paid!

A Reader Service number appears on each advertisement. Circle the corresponding numbers below for more info.

001	002	003	004	005	006	007	008	009
010	011	012	013	014	015	016	017	018
019	020	021	022	023	024	025	026	027
028	029	030	031	032	033	034	035	036
037	038	039	040	041	042	043	044	045
046	047	048	049	050	051	052	053	054
055	056	057	058	059	060	061	062	063
064	065	066	067	068	069	070	071	072
073	074	075	076	077	078	079	080	081
082	083	084	085	086	087	088	089	090
091	092	093	094	095	096	097	098	099
100	101	102	103	104	105	106	107	108
109	110	111	112	113	114	115	116	117
118	119	120	121	122	123	124	125	126
127	128	129	130	131	132	133	134	135
136	137	138	139	140	141	142	143	144
145	146	147	148	149	150	151	152	153
154	155	156	157	158	159	160	161	162
163	164	165	166	167	168	169	170	171
172	173	174	175	176	177	178	179	180
181	182	183	184	185	186	187	188	189
190	191	192	193	194	195	196	197	198
199	200	201	202	203	204	205	206	207
208	209	210	211	212	213	214	215	216
217	218	219	220	221	222	223	224	225
226	227	228	229	230	231	232	233	234
235	236	237	238	239	240	241	242	243
244	245	246	247	248	249	250	251	252
253	254	255	256	257	258	259	260	261
262	263	264	265	266	267	268	269	270
271	272	273	274	275	276	277	278	279
280	281	282	283	284	285	286	287	288
289	290	291	292	293	294	295	296	297
298	299	999						

Circle 999 to start a 12 month subscription at the price of $29.97

Dr. Dobb's Journal of Software Tools

January 1986 #111 **Expiration Date: April 30, 1986**

Name ______________________ **Title** ______________

Company ____________________ **Phone** ______________

Address ____________________________________

City/State/Zip ________________________________

Please circle one letter in each category:

I. My work is performed:
A. for in-house use only.
B. for other companies.
C. for end users/retailers.
D. in none of the above areas.

II. My primary job function:
A. Software Project Mgmt/Spvr
B. Hardware Project Mgmt/Spvr
C. Computer Consultant
D. Corporate Management
E. Other

III. My company department performs:
A. software development.
B. computer system integration.
C. computer manufacturing.
D. computer consulting.
E. computer research.
F. none of the above.

IV. This inquiry is for:
A. a purchase within 1 month.
B. a purchase within 1 to 6 months.
C. product information only.

V. Corporate Purchase Authority:
A. Final Decision-maker
B. Approve/Recommend
C. No Influence

VI. Personal Computer Users at my Jobsite:
A. 10,000 or more
B. 500 to 9,999 499
C. 100 to
D. 10 to 99
E. less than 10

VII. On average, I advise others about computers:
A. more than once per day.
B. once per day.
C. once per week.
D. less than once per week.

VIII. In my job function, I:
A. design software and/or write code.
B. design software.
C. write code.
D. don't design software or write code.

Product Information

Free!

Because of the conditional assemblies in COM, I needed to place some labels on lines by themselves, particularly MLOOP, the main looping point of the opcode simulation. Though the assembler allows this (provided you put a colon after the label), I was unable to make the label global using the ".globl" directive. However, if you do something like this:

```
        .globl mloop
    mloop:
~ ~ mloop:      ; Why? I don't know!
```

you can get the declaration to work. This trick was found by examining the code produced by the C compiler. I believe that if the label is in uppercase you don't need this extra statement. Note that

```
        .globl mloop
mloop:  equ *
```

won't work because of the problem with "equ" described earlier.

The "offset" directive is extremely useful for generating data storage areas to be used with indexing. It would have been useful in COM, but it doesn't work. "Equ" must be used, and the programmer must count bytes in the storage areas to make sure nothing overlaps. Grrrrr. "Offset" wasn't accepted by the CP/M-68K Release 1.1 assembler. The Release 1.2 assembler works just fine—but neither linker will accept symbols defined in offset sections. Another "feature" to fix. Doesn't DRI test anything?

Another trick that may be useful in 68000 assembler programming is using the ".opd" directive to generate your own opcodes. These are particularly useful as stand-ins for less meaningful ".dc.w constant" sequences when generating data tables, although you may define instructions also. COM uses this trick to define the 68000's BDOS and BIOS trap instructions. Look at the file AS68INIT for more examples of its use. The only restriction for use is that the new opcode must follow the addressing rules of one of the existing 68000's instructions. Oh, for a true macro assembler. . . .

The C preprocessor, CP68, may be used as a poor man's macro assembler. If the assembly file (let's say TEST.MAC) looks something like this:

```
A>COMT TEST
Start trace at >5
Stop trace at >ff02
-AF-  -BC-  -DE-  -HL-  -SP-  -S0-  -S1-  -S2-  -S3-  -PC-  -op-
2300  4509  0200  0000  FFFE  0000  0000  0000  0000  0005   C3  JMP  FF00
-AF-  -BC-  -DE-  -HL-  -SP-  -S0-  -S1-  -S2-  -S3-  -PC-  -op-
2300  4509  0200  0000  FFFE  0000  0000  0000  0000  FF00   76  HLT
Printed by BDOS #9
-AF-  -BC-  -DE-  -HL-  -SP-  -S0-  -S1-  -S2-  -S3-  -PC-  -op-
2300  4509  0200  0000  FFFE  0000  0000  0000  0000  FF02   C9  RET
A>
```

Table 2

8080 SIMULATOR
(Continued from page 81)

```
#include  "MACRO.H"
label     testmac(1,2,3)
          end
```

and the file MACRO.H looks like this:

```
* File produced by using CP68 as a
  macro
* assembly preprocessor on a .MAC
  file.

#define testmac(x,y,z) dc.w x \
          .dc.w y \
          .dc.w z
```

and if the files are processed like this:

```
A>CP68 TEST.MAC TEST.S     ; CP/M 1.1
```

or

```
A>CP68 -P TEST.MAC TEST.S ; CP/M 1.2
```

then the output file (TEST.S) will look like this:

```
* File produced by using CP68 as a
  macro
* assembly preprocessor on a .MAC
  file.

<cr>
<cr>
<cr>
label     .dc.w 1
          .dc.w 2
          .dc.w 3
          end
```

The <cr> lines are additional blank lines, one for each of the lines of a #define in the .H file.

Notes

The principal reference for the 8080 model was the *MCS-80/85 Family User's Manual* by Intel Corporation.

This program is released to the public domain with the stipulation that it be used for noncommercial purposes and that appropriate credit be given in any upgrade.

DDJ

(Listings begin on page 104)

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 5.**

C CHEST LISTING

(Text begins on page 18)

```
#include <stdio.h>
#include <dir.h>
#include <process.h>
#include <errno.h>
#include <fcntl.h>
#include <signal.h>
#include <dos.h>

/*  SH.C:   A shell for MSDOS.
 *
 *        Copyright (C) 1985 Allen I. Holub. All rights reserved.
 *
 *-----------------------------------------------------------------
 *
 *      This file contains an MSDOS shell that operates in conjunction
 *      with command.com. It has several built-in commands (see below).
 *      It recognizes several semi-colon seperated commands on a line
 *      & does wild card expansion. It can take commands interactively
 *      or from the command line (but command.com will intercept the
 *      re-direction if you have any on the command line). It supports
 *      simple batch files and will expand several $ arguments.
 *
 *      It does not support redirection yet.
 *
 *-----------------------------------------------------------------
 *   Usage:     sh [-cvx] <args>
 *
 *  Invocation:
 *
 *  sh
 *  sh -i               shell entered in interactive mode. If -i is given
 *                      any command line arguments that follow will be
 *                      assigned to $0 $1 etc. $0 will point at the
 *                      leftmost argument following the -i.
 *
 *  sh -c <string>      commands are read from string. If several strings
 *                      are present they are concatanated together before
 *                      execution. The 'c' may be upper or lower case.
 *
 *  sh <file> args...   commands are taken from <file>. Args are expanded
 *                      to correspond to $0 $1 etc inside the file. For
 *                      DOS compatability %0 %1 etc are also recognized.
 *                      $0 will be <file> itself.
 *
 *  sh -q               Strip quotes from quoted argument strings. Usually
 *                      they're left in so that a spawned process can
 *                      assemble its argv correctly.
 *  sh -v               Print input lines to the shell as they are read
 *                      (same as "set verbose=1").
 *  sh -x               Print lines as they are executed. (same as
 *                      saying "set echo=1" in the shrc.bat file.
 *
 *-----------------------------------------------------------------
 * Enviornment variables:
 *      CMDLINE    (set)     Holds the complete, 2048 byte, command line
 *                           that can't be passed via MSDOS.
 *      PROMPT     (used)    Defines the prompt string. Any character
 *                           or $<arg> may be used. Default prompt is
 *                           [$s:$p].
 *      SHLEV      (set)     Current shell level (0 is outermost). Remember
 *                           that batch files are executed in their own
 *                           shell.
 *      SWITCHAR  (used)     Use first character in line to designate
 *                           command line switches.
 *-----------------------------------------------------------------
 * Files:
 *      /shrc.bat        Executed every time a shell is created.
 *      /login.bat       Executed when a level 0 shell is created.
 *      /logout.bat      Executed when "logout" command is executed
 *-----------------------------------------------------------------
 * Built in commands:
 *      alias    [name [wordlist]]
 *      cd       <directory name>
 *      exit
 *      history
 *      logout
 *      pwd
 *      rem
 *      setenv  <name> [=] <value>
 *      set     [{cmd|echo|verbose|$arg} [= val]]
 *      shift
 *      unalias <name>
 *      unset   <name>
 *      !  !!  !<num>  !<pat>
 *      ^  ^^  ^<num>  ^<pat>
 *      !> [name]
 *      !< [name]
 *      #
 *-----------------------------------------------------------------
 * Pre-defined shell variables (may use either % or $) :
 *
 *      $<num>  one argument in batch file. $0 is file name.
 *      $*      All the $<num>s concatantated with spaces between them.
 *      $p      Full path name of current directory.
 *      $!      Current History number.
 *      $s      Nesting level of current shell. 0 is the outermost
 *
 *-----------------------------------------------------------------
 * Special characters:
 *      ;       Used to delimit two commands on one line.
 *      *?      Name containing these is expanded to matching directory
 *              entry.
 *
 *      Special characters aren't recognized in quoted strings or
 *      when preceeded by a backslash.
 *      All lines with # in the left-most column are ignored.
```

```
 *-----------------------------------------------------------------------
 */

extern int       access   (char*, int        ); /* in stdio library     */
extern int       bdos     (int,   int, int   );
extern int       chdir    (char*             );
extern int       errno;
extern FILE      *fopen   (char*, char*      );
extern int       fseek    (FILE*, long, int  );
extern long      ftell    (FILE*             );
extern char      *getenv  (char*             );
extern char      *getcwd  (char*, int        );
extern char      *malloc  (unsigned          );
extern int       putenv   (char*             );
extern int     ( *signal  (int,int(*)()) )()  ;
extern int       strlen   (char*             );
extern char      *strpbrk (char*,   char*    );

                                                /* source is in:        */
extern char      *cpy      (char*, char*      ); /* /src/tools/cpy.c     */
extern void      del_dir   (DIRECTORY *       ); /* /src/tools/dir.c     */
extern void      dir       (char*, DIRECTORY* ); /* /src/tools/dir.c     */
extern char      *efgets   (char*, int,  FILE*); /* /src/tools/efgets.c  */
extern DIRECTORY *mk_dir   (int               ); /* /src/tools/dir.c     */
extern char      *next     (char**,  int, int ); /* /src/tools/next.c    */
extern char      *skipto   (int,   char*, int ); /* /src/tools/skipto.c  */
extern char      *strsave  (char*             ); /* /src/tools/strsave.c */
extern int    unargv (int,char**,char*,int,int); /* /src/tools/unargv.c  */

extern void     unsetvar( char* );               /* ./var.c:             */
extern int      setvar( char*, char* );
extern void     printalias();
extern void     printvars();
extern int      getvar(char**, char**, int*);

extern void     print_hist(FILE*);               /* ./hist.c             */
extern int      get_hnum();
extern void     history( char*, int);

/*----------------------------------------------------------------------*/

#ifdef DEBUG
        static  int      Lev = -1;
#       define TRACE(p)     printf("%*s{ entering %s\n" ,++Lev * 4, "", p)
#       define END_TRACE(p) printf("%*s} exiting  %s\n" ,Lev-- * 4, "", p)
#       define DIAG(f,a)    printf(f,a)
#else
#       define TRACE(p)
#       define END_TRACE(p)
#       define DIAG(f,a)
#endif


#ifdef STR_CMDS
#       define PSTR(subr,str) printf("%s <%s>\n", subr, str);
#else
#       define PSTR(subr,str)
#endif

/*----------------------------------------------------------------------*/

#define VER         "1.0"      /* Version number                        */

#define DOSMAXLINE 127         /* Maximum line number permitted by DOS  */
#define MAXLINE    (2048+1)    /* Largest input command line in bytes +1 */
#define MAXDIR     128         /* Largest number of objects on cmd line */
#define CNTL_Z     ('Z'-'')
#define COMMENT    '#'         /* Deliniates comments                   */

#define ISQUOTE(c)       ((c)=='"' || (c)=='\'')
#define ISWHITE(c)       ((c)==' ' || (c)=='\t')
#define SKIPWHITE(p)     while( ISWHITE(*p) ){ p++; }
#define ISVAR(c)         ( (c)=='$' || (c)=='%' )

                         /* Possible modes in which shell can operate  */
#define FILEMODE    0    /* Get input from a file                      */
#define INTERACTIVE 1    /* Get input interactively from stdin         */
#define COMMAND     2    /* Get input from the command line            */

#define PMODE() ( Mode==COMMAND ? "COMMAND" : \
                  (Mode==FILEMODE ? "FILE" : "INTERACTIVE")  )

/*-----------------------------------------------------------------------
 *      Token definitions for built-in commands.
 */

typedef enum
{
        ALIAS,
        CD,
        CMD,
        EXIT,
        HISTORY,
        LOGOUT,
        PWD,
        REM,
        SET,
        SETENV,
        SHIFT,
        UNALIAS,
        UNSET
} TOKEN;

/*----------------------------------------------------------------------
 *      Global variables:
 */

static char** Numv ;          /* Vector array for expanding $<num> vars  */
static int    Numc  = 0;      /* count of valid entries in the above     */
static char   Ibuf[MAXLINE]; /* Input buffer                             */

```

(Continued on next page)

C CHEST LISTING

(Listing Continued, text begins on page 18)

```
static int Cmd          = 1 ; /* Generate CMDLINE env with spawned proc */
static int Switchar     ='-'; /* Designates command line switches        */
static int Verbose      = 0;  /* Print input lines as they're read  -v  */
static int Echo         = 0;  /* Print commands as they're executed -x  */
static int Noquotes     = 0;  /* Strip " or ' from quoted args      -q  */
static int Shlev        = -1; /* Nesting level of the current shell     */
static char *Filename   = 0;  /* Full path name of file specified in    */
                              /* Filemode input.                        */

/*----------------------------------------------------------------------
 * Set up input mode for file mode processing. Note that Last_posn and
 * Ateof are used by file_input().
 */

char            *file_input();
static int      Mode            = FILEMODE   ;
static char     *(*Ifunct)()    = file_input ;
static long     Last_posn       = 0L ;
static int      Ateof           = 0  ;

reset_fileinput()
{
        Last_posn = 0L          ;
        Ateof     = 0           ;
        Mode      = FILEMODE    ;
        Ifunct    = file_input  ;
}

/*----------------------------------------------------------------------
 *      Making isdigit into a subroutine makes processing marginally
 *      easier in exp_vars (below).
 */

digit(c)
{
        return( '0'<= c  &&  c <= '9' );
};

/*----------------------------------------------------------------------
 * Input functions:     interactive_input()     Gets input from keyboard
 *                      command_input()         Gets input from cmd line
 *                      file_input()            Gets input from a file.
 *
 *      Only one of the three will be used depending on the way that
 *      the shell was invoked. All three return 0 on end of input, a
 *      pointer to the beginning of the current input line on success.
 *      The input line will have been loaded into the global array
 *      Ibuf, which is assumed to be dimensioned to MAXLINE characters.
 */
/*--------------------------------------------------------------------*/

char    *interactive_input()
{
        register char   *rval = NULL;

        TRACE("interactive_input");

        *Ibuf = 0;
        if( efgets(Ibuf, MAXLINE, stdin) )
                rval = Ibuf ;

        END_TRACE("interactive_input");
        return rval;
}

/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/

char    *command_input()
{
        static  int     have_been_called = 0 ;
        register char   *rval            = NULL ;

        TRACE( "command_input" );

        if( !have_been_called )
        {
                unargv( Numc, Numv, Ibuf, MAXLINE, ' ' );
                rval = Ibuf ;
                have_been_called++;
        }

        END_TRACE( "command_input" );
        return rval;
}

/*- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -*/

char    *file_input()
{
        /*      Get input for file mode. This kludge is required because
         *      a child process will close any open files when it exits.
         *      Consequently we open the file, get a line, remember
         *      the position within the file, and then close the file
         *      on each call. On the next call we'll return to the
         *      position we remembered in the previous call.
         */

        register char   *rval     = NULL ;
        register FILE   *fp;

        TRACE( "file_input" );

        if( !Ateof )
        {
                if( !(fp = fopen(Filename, "r")) )
                    fprintf(stderr,"Sh: Can't open batch file <%s>\n",
                                                        Filename);
```

```
            else if( !fseek(fp, Last_posn, 0) )
            {
                /* Get a line from the buffer. Note that if the
                 * file doesn't have it's last line terminated
                 * with a carriage return, efgets will return
                 * true even though we're at end of file
                 * thus the call to feof.
                 */

                *Ibuf = 0;
                rval  = efgets(Ibuf, MAXLINE, fp) ? Ibuf : NULL ;
                if( !(Ateof = feof(fp)) )
                        Last_posn = ftell( fp ) - 1;
                fclose( fp );
            }
        }

        END_TRACE( "file_input" );
        return( rval );
}

/*----------------------------------------------------------------------*/

int     has_wild( bp )
register char   *bp;
{
        /*      Return true if the string has a * or ? in it
         */

        for(; *bp ; ++bp)
                if( *bp == '*' || *bp == '?' )
                        return 1;

        return 0;
}

/*----------------------------------------------------------------------*/

int     strip( src )
register char       *src;
{
        /*      Take care of special characters in a string (* and ?).
         *
         *      Copy src onto itself, stripping out back-slashes. If a
         *      * or ? which isn't preceeded by a backslash is found
         *      return 1, else return 0. If the first character in the
         *      string is a quote then * and ? aren't special.
         */

        register char   *dest   = src;
        int             special = 0;

        TRACE("strip");

        while( *src )
        {
                if( *src == '\\' )
                {
                        /* Copy the char. following the \ into
                         * dest and then, if we aren't at end of
                         * string, advance src to point past the
                         * escaped character
                         */

                        *dest++ = *++src;
                        if( *src )
                                ++src;
                }
                else if( ISQUOTE(*src) )
                {
                        /*  Copy a quoted string verbatum, removing
                         *  the quotes if Noquotes (-q) was given on
                         *  the command line.
                         */

                        if( Noquotes )
                                ++src ;

                        while( *src && !ISQUOTE(*src) )
                        {
                                if( src[0] == '\\' && src[1] )
                                        *dest++ = *src++;

                                *dest++ = *src++ ;
                        }

                        if( *src )  /* Then *src is a quote */
                        {
                                if( Noquotes )
                                        src++;
                                else
                                        *dest++ = *src++ ;
                        }
                }
                else
                {
                        /* Just do the copy. Set special to true
                         * if we copy a special character.
                         */

                        if( *src == '*' || *src == '?' )
                                special = 1;

                        *dest++ = *src++;
                }
        }

        *dest = '\0';

        END_TRACE("strip");
```

(Continued on next page)

C CHEST LISTING

(Listing Continued, text begins on page 18)

```

        return( special );
}

/*----------------------------------------------------------------------*/

char    *nextarg( lp )
char    **lp;
{
        /*      Get the next, space delimited, argument from the string
         *      pointed to by *lp. Return a pointer to the argument and
         *      update *lp to point past it. Leading white space is
         *      skipped.
         */

        register char   *start, *line = *lp;

        TRACE("nextarg");
        SKIPWHITE( line );

        if( !*line  )
            start = (char *)0 ;
        else
        {
            start = line++;

            line = skipto( ISQUOTE(*start) ? *start : ' ', line , '\\');

            if( ISQUOTE(*line) )
                line++;

            if( *line )
                *line++ = '\0' ;

            *lp = line;
        }

        END_TRACE("nextarg");

        return start;
}

/*----------------------------------------------------------------------*/

int     exp_dir( buf, maxcount )
char    *buf;
{
        /*      Remake buf, expanding any wild card characters into
         *      their proper names. That is, if buf containg the string:
         *      "foo * bar" and the current directory contains the
         *      files A, B and C then on exit, buf will point
         *      at the string "foo a b c bar". Expanded entries are
         *      sorted. Return 0 if a wild card that couldn't be
         *      expanded was found, else return 1.
         */

        register DIRECTORY *dp = 0 ;
        register char      *arg, *sbuf, *p ;
        int                i, rval = 1;

        TRACE("exp_dir");

        if( !(dp = mk_dir(MAXDIR)) )
                goto abort;

        dp->files = 1;          /* Get all files                        */
        dp->dirs  = 1;          /* and all directories                  */
        dp->path  = 1;          /* and prepend the path name if given   */
        dp->sort  = 1;          /* the list should be sorted            */

        /*      Get the arguemts from the input src, one at a time,
         *      putting them into the dirv array of a DIRECTORY structure
         *      (one argument per dirv entry).
         *
         *      If the argument has special characters (* or ? not
         *      preceded by a \ or enclosed by quotes) then expand to a
         *      directory using dir(), otherwise just put the argument
         *      into the dirv array directly.
         *
         *      Buf will grow larger if anything is expanded but it
         *      won't get larger than maxcount.
         */

        for( sbuf = buf;  (arg = nextarg(&buf))  &&  dp->maxdirs > 0; )
        {
                if( strip(arg) )
                {
                        i = dp->maxdirs;

                        dir( arg, dp );

                        if( dp->maxdirs == i )
                        {
                                /*  dir() didn't do anything    */

                                fprintf(stderr,"Sh: Can't expand <%s>\n",arg);
                                rval = 0;
                                goto abort;
                        }

                }
                else
                {
                        /* Add a command to dirv. First malloc space
                         * for it. Then copy it the arg into the
                         * malloc()ed space & put it into dirv.
                         * We use malloc in order to make it easier
                         * to free the space used by the DIRECTORY
```

```
                         * structure.
                         */

                        if(!( p = malloc( strlen(arg)+1 )  ))
                        {
                                fprintf(stderr, "Sh: out of memory !!\n");
                                goto abort;
                        }

                        strcpy( p, arg );

                         *( dp->lastdir )++ = p ;
                        --( dp->maxdirs );
                        ++( dp->nfiles  );
                }
        }

        i = unargv(dp->nfiles+dp->ndirs, (char **)dp->dirv,sbuf,maxcount,' ');

        if( dp->maxdirs <= 0 || i >= maxcount-1 )
                fprintf(stderr, "Sh: command line too large, truncating\n");

abort:  if( dp )
                del_dir( dp );

        END_TRACE("exp_dir");
        return( rval );
}

/*----------------------------------------------------------------------*/

char    *search( fname, ext )
char    *fname, *ext;
{
        /*  Search for fname.ext in the current PATH. Return a pointer
         *  to the full path name if you find it (0 if you don't).
         */

        static   char    pathname[80], pbuf[129] ;
        register char    *p ;
        char             *paths ;

        /*      Assemble the pathname by concatanating fname and ext
         */

        TRACE("search");

        if( strpbrk(fname, ".") )           /* If file name already has an  */
                ext = "";                   /* extension don't add another. */

        sprintf( pathname, "%0.32s.%0.3s", fname, ext );

        if( access(pathname, 04) < 0 )
        {
                /*      The file doesn't exist in the current directory.
                 *      If fname contains the characters \ or / or if
                 *      the PATH enviornment isn't set, return a NULL,
                 *      else search for it along the path. strpbrk() is
                 *      a microsoft and Lattice library function. It'll
                 *      return true if fname contains a / or a \.
                 */

                if( strpbrk(fname,"\\/")  ||  !(p = getenv("PATH")) )
                        *pathname = '\0';
                else
                {
                        strncpy( (paths = pbuf), p, 129 );

                        while( p = next( &paths, ';', -1 ))
                        {
                            sprintf(pathname, "%0.50s\\%0.20s.%0.3s",
                                                p, fname, ext);

                            if( access( pathname, 04 ) >= 0 )
                                break;
                            else
                                *pathname = '\0';
                        }
                }
        }

        END_TRACE("search");

        return( *pathname ? pathname : NULL );
}

/*----------------------------------------------------------------------*/

int     execute( buf )
char    *buf;
{
        /*      Execute a command. Return the programs exit status or
         *      -1 if the program didn't execute. This routine assumes
         *      that buf is at least DOSMAXLINE characters long.
         */

        register int    rval = 0;
        register char   *name;
        static   char   envstr[MAXLINE+8] = "CMDLINE=";

        TRACE("execute");
        PSTR( "execute(1) buf=", buf );

        /*      Create the CMDLINE environment variable if Cmd it true
         *      (it can be set false with a "set cmd=0." If cmd is false
         *      then generate a "CMDLINE=" with no argument. This is
         *      necessary because MSDOS doesn't support a unset command.
         */

        if( Cmd )
```

(Continued on next page)

C CHEST LISTING

(Listing Continued, text begins on page 18)

```
                strncpy( &envstr[8], buf, MAXLINE );
        else
                envstr[8] = '\0' ;

        setenv( envstr , 0 );

        /*      Truncate the command line itself (not the enviornment
         *      variable, at 127 characters and then echo it to the
         *      screen if Echo is set. Then extract the program name
         *      portion of the string (with the next() call).
         */

        buf[DOSMAXLINE-3] = '\0';

        if( Echo )
                puts( buf );

        name = next(&buf, ' ', -1 );


        /*      Now try to spawn a new process. Suspend the shell until
         *      task returns.
         */

        if( (rval = spawnlp(P_WAIT, name, name, buf, NULL)) < 0 )
        {
                /*      If we can't find a .com or .exe file then
                 *      see if we can find a batch file with ole
                 *      right name. Otherwise print an error message
                 *      Note that we have to modify the CMDLINE
                 *      enviornment string so that the leftmost
                 *      argument (will be argv[0] in the child
                 *      process) is the string "sh"
                 */

                if( errno == ENOENT )
                {
                        sprintf(envstr, "CMDLINE=sh %s %s", name, buf);
                        rval = spawnlp(P_WAIT, "sh", "sh", name, buf, NULL);
                }
                if( rval < 0 )
                {
                        printf("sh: Can't execute %s %s", name, buf);
                        perror(" ");
                }
        }

        END_TRACE("execute");
        return rval;
}

/*----------------------------------------------------------------------*/

int     exp_vars( dest, src, maxcount, mode )
char    *dest, *src;
{
        /*      Copy src into dest, expanding shell variables as
         *      appropriate. Shell variables all have a $ or a % as their
         *      first character. The global pointers Numv and Numc keep
         *      track of arguments for $<num> variables.
         *
         *      If mode == 1      aliases are expanded
         *      If mode == 2      $args are expanded
         *      If mode == 3      both are expanded.
         *
         *      return the size of the expanded string.
         *      A mode 1 or 3 call will return with the high bit of src
         *      set.
         */

        register int    num;
        register char   *p ;
        char            *start_dest = dest;

        TRACE("exp_vars");
        DIAG("exp_vars, input = <%s>\n", src );


        /*      Expand aliases. First, remember the original start of
         *      the target array. Then, set the high bit of *src so that
         *      getvar will look for an alias. Then actually expand it,
         *      Then clear the high bit of the first character of dest
         *      (which will be set if no alias was found).
         *      There is a second-order recursion here in that getvar()
         *      makes a mode 2 exp_vars call.
         */

        if( mode & 1 )
        {
                *src |= 0x80;
                getvar( &src, &dest, &maxcount );
                DIAG("exp_vars, expanded aliases <%s>\n", start_dest);
        }


        /*      Now, expand shell variables. If we see and escaped
         *      character copy both the \ and the character to dest.
         *      else if the character isn't a shell variable (doesn't
         *      have a leading $ of %) just copy it. Else, try to
         *      expand the shell variable.
         */

        while( *src  &&  maxcount > 0 )
        {
                if( *src == '\\' && src[1] )
                {
                        /* If the character following the \ is a
```

```
                         * $ of % then strip the backslash and copy
                         * the $ or % to the dest array. Otherwise
                         * copy both the \ and the character that
                         * follows.
                         */

                        src++;          /* Skip past the \      */

                        if( !ISVAR(*src) )
                        {
                                if( --maxcount <= 0 )
                                        break;
                                *dest++ = '\\' ;
                        }

                        if( --maxcount <= 0 )
                                break;
                        *dest++ = *src++;
                }
                else if( !ISVAR( *src )  ||  !(mode & 2) )
                {
                        /* Either character is just a normal character
                         * (not a shell variable) or we're told to not
                         * expand shell variables.
                         */

                        *dest++ = *src++;
                        if( --maxcount <= 0 )
                                break;
                }
                else if( digit( *++src ) )
                {
                        /*      Expand a $<num> arg. first extract the
                         *      <num> from source, then copy the correct
                         *      vector out of the Numv array.
                         *      If the Numv entry is NULL, don't
                         *      put anything in the dest array.
                         */

                        for( num = 0; digit(*src); )
                                num = (num * 10) + (*src++ - '0');

                        if( num < Numc )
                            for( p = Numv[num]; *p ; *dest++ = *p++ )
                                if( --maxcount <= 0 )
                                    break;
                }
                else
                {
                        /* We've found a $ not followed by a number */

                        switch( *src++ )
                        {
                        case '*':
                                num = unargv(Numc,Numv,dest,maxcount,' ');
                                dest     += num;
                                maxcount -= num;
                                break;

                        case 'p':
                                getcwd( dest, maxcount );
                                for( ;*dest ; ++dest, --maxcount )
                                        if( *dest == '\\' )
                                                *dest = '/';
                                break;

                        case '!': num = get_hnum();  goto skip;
                        case 's': num = Shlev;
                        skip:     if( maxcount > 5 )
                                  {
                                        sprintf( dest, "%d", num );
                                        for(;*dest ; ++dest, --maxcount)
                                                ;
                                  }
                                  break;

                        default:
                                --src;
                                getvar( &src, &dest, &maxcount );
                                break;
                        }
                }

                *dest = '\0' ;
        }

        *start_dest &= 0x7f ;

        DIAG("exp_vars: on return <%s>,", start_dest );
        DIAG(" returning %d\n",           dest - start_dest );

        END_TRACE("exp_vars");

        return( dest - start_dest );
}

/*----------------------------------------------------------------------*/

prompt()
{
        /* Print a prompt using the PROMPT enviornment variable
         * Return true.
         */

        char            buf[50];
        register char   *p;

        if( Mode == INTERACTIVE  ||  Echo )
        {
                if( !(p = getenv("PROMPT")) )
                        printf( "[%d:%d] ", Shlev, get_hnum() );
```

(Continued on next page)

C CHEST LISTING

(Listing Continued, text begins on page 18)

```
                else
                {
                        exp_vars( buf, p, 50, 2 );
                        printf( buf );
                }
        }

        return 1;
}

/*----------------------------------------------------------------------*/

int     docmd( cmd )
char    *cmd ;
{
        /*      Do one command from line.
         *      Return an exit status (or -1 on error).
         */

        register DIRECTORY  *dp;
        register int        rval = -1;

        TRACE("docmd");
        PSTR( "docmd(1) cmd=", cmd );

        /* If ther're no wild cards in cmd then just strip backslashes
         * and quotes. Else, expand the wild cards ( exp_dir will strip
         * backslashes etc.) Then execute the command.
         */

        if( !has_wild(cmd) )
                strip(cmd);             /* Just strip backslashes */

        else if( !exp_dir(cmd, MAXLINE) )
                goto abort;

        rval = execute(cmd) ;

abort:  END_TRACE("docmd");
        return( rval );
}

/*----------------------------------------------------------------------*/

rcopy( dest, src, maxcount )
register char   *dest ;
char            *src  ;
{
        register char   *tbuf;
        char            *tail;
        char            *sd = dest;

        DIAG("rcopy (top of proc): src  = <%s>\n", src );

        if( !*src || maxcount <= 0 )
                return;

        if( !(tbuf = malloc( maxcount )) )
                fprintf(stderr,"Sh: out of memory\n");
        else
        {
                /* 1) expand the src buffer into tbuf
                 * 2) advance tail to point past the next ; and replace
                 *    the ; with a null.
                 * 3) copy everything up to the tail to dest and add a
                 *    ; to dest if the tail is non-null.
                 * 4) repeat this process using the tail as source.
                 */

        /*1*/   exp_vars( tbuf, src, maxcount , 3 );

                tail = tbuf;
        /*2*/   next( &tail, ';' , '\\');
                SKIPWHITE( tail );

        /*3*/   for(src = tbuf; *src && src-tbuf < maxcount; *dest++ = *src++)
                        ;

                if( (maxcount -= (src-tbuf))  && *tail )
                        *dest++ = ';' ;

                *dest   = '\0';

                DIAG("rcopy (before recursive call): dest = <%s>\n", sd );

        /*4*/   rcopy(dest, tail, maxcount);
                free(tbuf);

                DIAG("rcopy (after  recursive call): dest = <%s>\n", sd );
        }
}

/*----------------------------------------------------------------------*/

char    *next_cmd()
{
        /*      Get a line from input, split off one command, and
         *      then return a pointer to it (or NULL if at end of
         *      input. History processing is done here too.
         *      Semicolons inside quoted strings or preceeded by a '\'
         *      do not seperate commands. Comments and blank lines
         *      are absorbed (ie. next_cmd won't return until it
         *      gets a real input line).
         */

        static   char   Cmdbuf[MAXLINE]; /* Should initialize to zeros */
```

```
        static  char    *src = Cmdbuf  ;
        register char   *p ;
        char            *tbuf;

        TRACE("next_cmd");

        DIAG("next_cmd: src is <%s>\n", src );

        while( !*src )
        {
                do
                {
                        prompt();

                        if( !(*Ifunct)() )
                        {
                                END_TRACE("next_cmd" );
                                return NULL;
                        }

                        DIAG("next_cmd: got <%s> from input\n", Ibuf );

                        src = Ibuf;
                        SKIPWHITE( src );
                }
                while( !*src || *src == COMMENT );

                history( src, MAXLINE - (src-Ibuf) );
                if( *src )
                {
                         rcopy( Ibuf, src, MAXLINE );
                         src = Ibuf;
                }
        }

        p = next( &src, ';' , '\\') ;

        DIAG("next_cmd: buffer <%s>\n", p    );
        DIAG("next_cmd: on next call buffer will be <%s>\n", src );

        strcpy( Cmdbuf, p );
        if( Verbose )
                printf("[sh %d input] <%s>\n", Shlev, Cmdbuf );

        DIAG("next_cmd: returning <%s>\n", Cmdbuf );
        END_TRACE( "next_cmd" );

        return( Cmdbuf );
}

/*--------------------------------------------------------------------*/

char    *errmsgs[] =
{
"sh             shell entered in interactive mode\n",
"sh %ci          enter interactive mode\n",
"sh %cc <string>  commands are read from string. If several strings\n",
"               are present they are concatanated together before\n",
"               execution. The 'c' may be upper or lower case.\n",
"sh file args... commands are taken from <file>. Args are expanded\n",
"               to correspond with $0 $1 etc inside the file.\n",
"               for DOS compatability %0 %1 etc are also recognized\n",
"sh %cq          Strip quotes from quoted argument strings. Usually\n",
"               they are left in so that a spawned process can\n",
"               assemble its argv correctly.\n",
"sh %cv          Print input lines to the shell as they are read.\n",
"sh %cx          Print lines as they are executed\n",
""
};

usage( c )
{
        /*      Print the usage error message.
         */

        register char   **pp;

        fprintf(stderr, "Sh: illegal argument <%c>: Usage is\n\n", c);
        for( pp = errmsgs; **pp ; fprintf(stderr, *pp++, Switchar) )
                ;
        exit( 1 );
}


/*-------------------------------------------------------------------*/

int             setargs( p )
register char   *p;
{
        /*      Set various global flags base on command line
         *      arguments. This routine will also be used by
         *      "set" which can't recognize the -c switch. So,
         *      ignore -c if c_enabled is false. p should point
         *      at the - on entry. Processing will terminate when
         *      a space or tab or end of string is found.
         */

        TRACE("setargs");

        if( *++p )
        {
             for(; *p ; p++)
             {
                switch( *p )
                {
                case 'c':
                case 'C':  Mode = COMMAND;              break;
                case 'q':  Noquotes = 1;                break;
                case 'v':  Verbose  = 1;                break;
                case 'x':  Echo     = 1;                break;
```

(Continued on next page)

C CHEST LISTING

(Listing Continued, text begins on page 18)

```
                        case 'i':  Mode     = INTERACTIVE;        break;
                        case ' ':
                        case '\t':                                goto abort;
                        default:   usage( *p );                   break;
                        }
                }
        }
abort:
        END_TRACE("setargs");
}

/*----------------------------------------------------------------------*/

doargs(argc, argv)
char    **argv;
{
        if( --argc <= 0 )                  /* Check for comand line args & */
        {                                  /* use interactive mode if none */
                Mode = INTERACTIVE;        /* are found.                   */
                Ifunct = interactive_input;
                return;
        }

        if( **(++argv) == Switchar )       /* There's at least one arg,    */
        {                                  /* skip to second arg and call  */
                setargs( *argv++ );        /* set args if the first char   */
                --argc;                    /* is a -. Then skip past the   */
        }                                  /* second arg too.              */

        Numv = argv ;
        Numc = argc ;

        if( argc <= 0  || !**argv )
        {
                fprintf(stderr,"Sh: missing file name, ");
                fprintf(stderr,"use -i for interactive input\n");
                exit(1);
        }
        else if( Mode == COMMAND )
        {
                Ifunct = command_input;
        }
        else if( Mode == FILEMODE  && !(Filename = search(*argv, "bat")))
        {
                fprintf(stderr,"Sh: can't find <%s>\n", *argv );
                exit(1);
        }
}

/*----------------------------------------------------------------------*/

setenv( env, allocate )
char    *env;
{
        /*      Set an enviornment. Env may be "name=contents" or the = may
         *      be a space.
         *      If allocate is true allocate space, otherwise acutally use
         *      env (which must be static) as the string.
         */

        register char   *p;

        /*      Look for either a space or a = in the string. If you
         *      find a space, replace it with an =.
         */

        for( p = env ; *p && *p != ' ' && *p != '=' ; p++ )
                ;
        if( *p == ' ' )
                *p = '=' ;

        /*      Now set the enviornment to the indicated value
         */

        if( p = allocate ? strsave( env ) : env )
                if( putenv(p) != -1 )
                        return;

        fprintf(stderr,"Sh: setenv failed, out of memory\n");
        if( allocate && p )
                free(p);
}

/*----------------------------------------------------------------------*/

set( str )
register char   *str;
{
        /* Set a shell variable, syntax is
         *      [whitespace] <name> [= <value>]
         */

        register int    i;
        char            *name;

        /*      Get the name, replacing the trailing blank or = with
         *      a null. Print variables if there is no name.
         */

        if( !*str )
        {
                printf("%-8s: %s\n", "echo",    Echo    ? "ON" : "OFF");
                printf("%-8s: %s\n", "cmd",     Cmd     ? "ON" : "OFF");
                printf("%-8s: %s\n", "verbose", Verbose ? "ON" : "OFF");
                printvars();
        }
        else
```

```
        {
                for(name = str; *str ;  ++str )
                        if( *str == '=' || *str == ' ')
                        {
                                *str++ = 0;
                                break;
                        }

                /*      Get the command tail, skipping past any =
                 *      or blanks to get there.
                 */

                while( *str == '=' || *str == ' ' )
                        ++str ;

                /*      Process the command:
                 */

                i = *str ? atoi(str) : 1 ;

                if(      !strcmp( "echo",     name))  Echo    = i;
                else if( !strcmp( "verbose", name))  Verbose = i;
                else if( !strcmp( "cmd",      name))  Cmd     = i;
                else      setvar(name, str);
        }
}

/*----------------------------------------------------------------------
 *      Alias support uses the same tables as shell varialbes. However,
 *      the top bit of the first character of the alias name is set
 *      to indicate its function.
 */

unalias( str )
char    *str;
{
        if( *str )
        {
                *str |= 0x80 ;
                unsetvar( str );
        }
}

alias( str )
char    *str;
{
        if( !*str )
                printalias();
        else
        {
                *str |= 0x80 ;
                set( str );
        }
}

/*----------------------------------------------------------------------*/

void    pwd()
{
        /*      Print working directory
         */

        char    nbuf[80];

        if( !getcwd( nbuf, 80 ) )
                fprintf(stderr, "sh: path too long to print.\n");

        printf("%s\n", nbuf );
}

/*----------------------------------------------------------------------*/

disk_present( id )
int     id;
{
        /*      Return true if there's a disk plugged into the
         *      indicated drive, else print an error message
         *      and return 1. This routine assumes that drive
         *      C has a disk in it so it will return true if id == 'c'
         *      or id == 'C' without checking.
         */

        register int    try = 5;        /* times to try to read disk    */
        register int    err ;
        union  REGS     regs;

        if( (id = toupper(id)) == 'C' || id == 'c' )
                return 1;

        regs.h.ah = 4;          /*    Service #4 (verify sec) */
        regs.h.al = 1;          /*    # of sectors            */
        regs.x.cx = 0;          /*    track # & sector #      */
        regs.h.dh = 0;          /*    head #                  */
        regs.h.dl = id - 'A';   /*    drive #                 */

        /*      Actually read the disk. Loop until we've gotten a timeout
         *      error (0x80) from dos try times.
         */

        do{
                err = int86(0x13, &regs, &regs) & 0xff;

        } while(  (err & 0x80)  &&  (--try >= 0) );


        if( err )
        {
            regs.h.ah = 0x0;            /* Recalibrate diskette system */
            int86(0x13, &regs, &regs);

            fprintf(stderr,"Cd: can't log on drive %c, ", toupper(id));
```

(Continued on next page)

C CHEST LISTING

(Listing Continued, text begins on page 18)

```
		fprintf(stderr,"DOS error code = 0x%02x\n"  , err        );
	}

	return( !err );
}

/*----------------------------------------------------------------------*/

void		cd( name )
register char	*name;
{
	/*	Change the current directory to the indicated name.
	 *	Log in a new disk if necessary. This routine is
	 *	a bit more sophisticated than DOS itself, in that
	 *	it checks if a disk exists before trying to
	 *	log it on. This checking is done using the
	 *	"Get diskette status" service of BIOS interrupt 0x13.
	 *	Get to the current directory on another disk by saying
	 *	"cd x:" Get to another directory on another disk by
	 *	saying "cd x:/dir/subdir/etc"
	 */

	if( *name && name[1] == ':' )
	{
		if( !disk_present(*name) )
			return;
		else
		{
			bdos( 0xe, toupper(*name) - 'A', 0);
			name += 2;
		}
	}

	if( *name  &&  chdir( name ) < 0 )
		fprintf(stderr, "sh: Can't find %s\n", name );
}

/*----------------------------------------------------------------------*/

shift()
{
	/*	Process the "shift" command (move all the $ args left
	 *	one notch).
	 */

	if( Numc )
	{
		--Numc;
		++Numv;
	}
}

/*----------------------------------------------------------------------*/

doenv()
{
	/*	Reads and initializes the various enviornment variables.
	 *	This routine should only be called once and it must be
	 *	called before Shlev is used and before command line
	 *	processing is done.
	 */

	static  char	sbuf[16];
	register char	*p;

	if( (p = getenv("SWITCHAR"))  &&  *p )
		Switchar = *p ;

	if( (p = getenv("SHLEV"))     &&  *p )
		Shlev = atoi(p);

	sprintf(sbuf, "SHLEV=%d", ++Shlev );
	setenv (sbuf);
}

/*----------------------------------------------------------------------*/

use_exit()
{
	/*	We get here on a SIGINT (^C) interrupt
	 */

	signal( SIGINT, use_exit );
	fprintf(stderr,"Use \"exit\" or \"logout\" to leave outer shell.\n");
}

/*----------------------------------------------------------------------*/

TOKEN	tokenize( buf )
register char	*buf;
{
	/*	This is an extremely primitive token recognizer that will
	 *	eventually be replaced with something more reasonable.
	 */

	if( !strcmp( "alias",	buf)) return ALIAS;
	if( !strcmp( "cd",	buf)) return CD;
	if( !strcmp( "exit",	buf)) return EXIT;
	if( !strcmp( "history",	buf)) return HISTORY;
	if( !strcmp( "logout",	buf)) return LOGOUT;
	if( !strcmp( "pwd",	buf)) return PWD;
	if( !strcmp( "rem",	buf)) return REM;
	if( !strcmp( "setenv",	buf)) return SETENV;
	if( !strcmp( "set",	buf)) return SET;
	if( !strcmp( "shift",	buf)) return SHIFT;
	if( !strcmp( "unalias",	buf)) return UNALIAS;
	if( !strcmp( "unset",	buf)) return UNSET;
```

(Continued on page 98)

INTRODUCING DATALIGHT C

The Datalight C Compiler for MSDOS is a full C with all K&R constructs, including bitfields, plus the version 7 extensions. Other features of the compiler are:

$60

- Produces object files (.obj) so just the MSDOS linker is required.
- Floating point performed with 8087 or automatic software floating.
- Over 100 compact library functions with source.
- Compatible with the Lattice C compiler.
- Runs on IBM-PC, and compatibles, running MSDOS 2.0 or later.
- Complete, easy-to-read users' manual with index.
- Highly optimized code for production quality programs.

Datalight

11557 8th Ave. N.E.
Seattle, Washington 98125
(206) 367-1803

Outside USA add $10 shipping. Washington State residents add 7.9% sales tax. VISA and MasterCard accepted.

IBM-PC, a trademark of IBM; MSDOS, a trademark of Microsoft Corp.; Lattice C, a trademark of Lattice Corp.

Circle **no. 203** on reader service card.

INDUSTRIAL PASCAL FOR THE 68000

If you're looking for a language to write real-time process control software, look no further. With the rising cost of labor, it is becoming critical that a high level language be used whenever possible. Find out why over 1400 companies have switched to OmegaSoft Pascal for their demanding applications.

OmegaSoft Pascal takes the Pascal framework and expands the basic data types, operators, functions, and memory allocation to fit the needs of real-time systems. These additions fit in the same structure as Pascal and enhance its usefulness without impairing the excellent readability, ease of maintenance, and structured design.

The compiler package includes the compiler, interactive symbolic debugger, relocatable macro assembler, linking loader, and screen editor. Source code is provided for the debugger, screen editor and runtime library.

Versions to run under the OS-9/68000 and VERSAdos operating systems are currently available to end-users and OEM's. End user price is $900 (domestic) or $925 (international). A version for CP/M-68K is available for OEM use, with OEM versions for UNIX type operating systems to follow.

T.M. OmegaSoft is a trademark of Certified Software Corporation. OS-9/68000 is a trademark of Microware. VERSAdos is a trademark of Motorola. CP/M-68K is a trademark of DRI. UNIX is a trademark of Bell Labs.

CERTIFIED SOFTWARE CORPORATION
616 Camino Caballo, Nipomo, CA 93444
Telephone: (805) 929-1395; Telex: 467013

Circle **no. 209** on reader service card.

MORE... VALUE and PERFORMANCE *with Mitek's* Relocatable Z80 Macro Assembler and Z80 Symbolic Debugger

NEW Z80 Symbolic Debugger

- Only $49.95 plus shipping.
- Screen oriented with a simultaneous display of instruction mnemonics, register, stack, and memory values.
- Breakpoints may be set on any combination of fixed memory address, register values and/or memory values.
- Uses Digital Research compatible SYM files.
- Supports Hitachi HD64180.

Relocatable Z80 Macro Assembler

- Only $49.95 plus shipping.
- 8080 to Z80 Source Code Converter.
- Generates Microsoft compatible REL files or INTEL compatible hex files.
- Compatible with Digital Research macro assemblers MAC & RMAC.
- Generates Digital Research compatible SYM files.
- Conditional assembly.
- Phase/dephase.
- Cross-reference generation.
- Full Zilog mnemonics.
- INCLUDE and MACLIB FILES.
- Separate data, program, common, and absolute program spaces.
- Supports Hitachi HD64180.
- Z80 Linker and Library Manager for Microsoft compatible REL files available as an add-on to Assembler.

ATTENTION Turbo Pascal Users:
Assembler will generate Turbo Pascal in-line machine code include files.

TO ORDER, CALL TOLL FREE: 1-800-367-5134, ext. 804
For information or technical assistance: (808) 623-6361

Specify desired 5¼" or 8" format. Personal check, cashier's check, money order, VISA, MC, or COD welcomed.

P. O. Box 2151
Honolulu, HI 96805
MITEK

PRICE LIST

Z80 Macro Assembler: $49.95
Assembler, Linker, and Library Manager: $95.00
Manual Only: $15.00
Z80 Symbolic Debugger: $49.95
Manual Only: $15.00
Assembler, Linker, Library Manager, and Debugger: $134.95
Include $5 for shipping and handling.

Z80 is a trademark of Zilog, Inc. MAC and RMAC are trademarks of Digital Research, Inc. Turbo Pascal is a trademark of Borland International, Inc.

Circle **no. 190** on reader service card.

C CHEST LISTING

(Listing Continued, text begins on page 18)

```
          return CMD;
}

/*----------------------------------------------------------------------*/

cmds()
{
          /*      Process commands using the current Ifunct. Continue till
           *      end of input is reached. If expand_vars is false then
           *      aliases and $args aren't expanded. This lets us read
           *      in the shrc.bat file unmolested
           */

          char              cmdbuf[9];      /* Place to put extracted command */
          char              *p, *start;
          register char     *cmd;
          register int      i;
          int               rval = 0;

          while( start = cmd = next_cmd() )
          {
                  DIAG("cmds: next_cmd returned <%s>\n", cmd );

                  /*
                   *       Strip the command name from the rest of the command
                   */

                  p = cmdbuf;
                  for( i = 8; --i >= 0 && *cmd && *cmd != ' ' ; *p++ = *cmd++)
                          ;
                  *p = '\0';

                  DIAG("cmds: partitioned <%s>", cmdbuf );
                  DIAG("+ <%s>\n", cmd );

                  SKIPWHITE(cmd);

                  /* If Echo is set then echo the command to standard
                   * output. Note that if i == CMD then the command
                   * will be echoed in execute() (called by docmd()).
                   */

                  if( (i = (int) tokenize(cmdbuf)) != (int) CMD  &&  Echo )
                          puts( start );

                  switch( i )
                  {
                  case ALIAS:     alias   ( cmd );                  break;
                  case CD:        cd      ( cmd );                  break;
                  case CMD:       rval = docmd(start);              break;
                  case HISTORY:   print_hist( stdout );             break;
                  case PWD:       pwd     (     );                  break;
                  case REM:                                         break;
                  case SET:       set     ( cmd );                  break;
                  case SETENV:    setenv  ( cmd );                  break;
                  case SHIFT:     shift   (     );                  break;
                  case UNALIAS:   unalias ( cmd );                  break;
                  case UNSET:     unsetvar( cmd );                  break;
                  default:        printf  ("Illegal token\n");      break;

                  case LOGOUT:
                          reset_fileinput();
                          if( access(Filename = "/logout.bat", 04) == 0 )
                                  cmds();

                          /* Fall through to EXIT case */

                  case EXIT:
                          goto exit;
                  }
          }
exit:
          return rval;
}

/*----------------------------------------------------------------------*/

main(argc, argv)
char      **argv;
{
          /*      Exit status is that of the most
           *      recently excuted program. In the case of a batch file
           *      the exit status will be that of the shell processing
           *      the batch file (ie. of the last program executed by
           *      that shell).
           */

          reargv(&argc, &argv); /* Get a long command line if one's there*/
          doenv();              /* Read various enviornment variables and
                                 * intialize Shlev, which holds the
                                 * current shell level.
                                 */

          /*      Process the two automatic batch file. /shrc.bat is
           *      executed every time a shell is created. /login.bat
           *      is only executed when a level 0 shell is created.
           */

          if( access(Filename = "/shrc.bat", 04) == 0 )
          {
                  cmds();
                  reset_fileinput();
          }

          if( Shlev == 0  &&  access(Filename = "/login.bat", 04) == 0 )
          {
                  cmds();
```

```
                reset_fileinput();
        }


        /*      Now process the command line arguments. Doargs will
         *      set the Echo, Verbose, Cmd and Mode variables as
         *      appropriate. If we're in an interactive, level
         *      0 shell, Call signal to prevent ^C from working on
         *      the shell itself and print a copyright notice.
         */

        doargs(argc, argv);             /* Process command line args    */

        if( Shlev == 0  &&  Mode == INTERACTIVE )
        {
                signal( SIGINT, use_exit );
                fprintf(stderr,"SH (ver %s) - Copyright (c) 1985, ", VER);
                fprintf(stderr,"Allen I Holub. All rights reserved.\n");
        }


        /*   Finally, process commands from the input source determined
         *   by the command line.
         */

        exit( cmds() );
}
```

End Listing

Shell listings continue next month

LISTING ONE (Text begins on page 26)

```
/*
        Declare the format of a macro table node.
*/

struct node {
        struct node * next; /* Pointer to next node. */
        int       nargs;              /* Number of args in macro. */
        char *    name;               /* Pointer to name of macro. */
        char *    text;               /* Pointer to replacement text of macro. */
};

/*
        Define the hash table used to access nodes of the macro table.
*/

#define MAC_PRIME 101

struct node * ht [MAC_PRIME];

/*
        Look up a symbol in the macro table.

        Output:   Z        if found, NZ if not found.
                           a0      pointer to text of symbol.
                           d0w     number of arguments (0-n), -1 if no arguments.
*/

#define hash_val d2
#define bp0 a1
#define bp1 a2

lookup(symbol)
register char * symbol;
{
        register struct node **bp0, *bp1;
        register int hash_val;

        /* Get the hash value of the symbol into hash_val. */
        hash(symbol);
        hash_val = d0;

        /* Point bp0 into the hash table. */
        bp0 = &ht;
        hash_val *= sizeof(struct node *);
        adda(hash_val, bp0);

        /* Search down the list of buckets hanging from the hash table. */
        for (bp1 = *bp0; bp1; bp1 = bp1 -> next) {
                str_eq(symbol, bp1 -> name);
                if (Z) {
                        /* Match. */
                        a0 = bp1 -> text;
                        d0 = bp1 -> nargs;
                        move(Z_BIT, ccr);
                        return;
                }
        }

        /* No match. */
        move(NZ_BIT, ccr);
}
```

End Listing One

LISTING TWO

```
ht:     ds.l          101

lookup:
        movem.l       a1/a2/d2, -(sp) ;function entry
        subq          #4, sp

        move.l        20(sp), (sp) ;get the hash value into hash_val
        jsr           hash
        move.l        d0, d2
        move.l        #ht, a1 ;point bp0 into the hash table
        mulu          #4, d2
        adda.l        d2, a1

;       search down the list of buckets

        move.l        (a1), a2 ;for(bp1 = * bp0;...;...)
        bra           _3
_1:
        move.l        6(a2), (sp) ;str_eq(symbol, bp1 -> name);
        move.l        20(sp), -(sp)
        jsr           str_eq
        addq          #4, sp
        bnz           _2                       ;if (Z)
        move.l        6(a2), a0 ;              a0 = bp1 -> text;
        move.w        4(a2), d0 ;              d0 = bp1 -> nargs;
        move          #4, ccr ;                move(Z_BIT, ccr);
        bra           _4                       ;       return;
_2:
        move.l        (a2), a2 ;for(...;...;bp1 = bp1 -> next)
_3:
        cmpa.l        #0, a2 ;for (...; bp; ...)
        bnz           _1
        move          #0, ccr ;move(NZ_BIT, ccr);
_4:
        addq          #4, sp ;function exit
        movem.l       (sp)+, a1/a2/d2
        rts
```

End Listings

(Text begins on page 44)

```
;
;               Terra Nova Communications multi-tasking kernel
;               Initialization and task-switcher
;
;               Note: this is not intended to be a complete listing.  It's only
;               a sample of some of the techniques used in our system.
;

                PSECT       Kernel

;               External symbols (defined in other code segments)
                EXTERN      VecTable,               ;vector table for hardware vector list
                            JMPTable,               ;jump table for system calls
                            JMPTabLen,              ;length of ju mp table in longwords
                            KernEnd,                ;end of kernel code item in heap
                            IOInit,                 ;our private I/O initialization routine
                            HeapInit,               ;our private heap initialization
                            SysInit,                ;system variable initializer
                            SysConMon,              ;entry point for system console
                                                    ;monitor task
                            HeapMunger,             ;entry point for heap munger task
                            DiskMunger              ;entry point for disk munger task

;               Entry points in this module (referenced from elsewhere)
                ENTRY       Start,                  ;primary entry point to boot our OS
                            ConSwitch,              ;main context switcher
                            ConSwSleep              ;alternate context switcher (puts
                                                    ;calling task to sleep)

;               Include files (mostly equates)
                INCLUDE     SysEqu                  ;contains the low-memory absolute
                                                    ;address equates (jump table, etc)
                INCLUDE     HeapDef                 ;defines the heap data structure
                INCLUDE     SysIO                   ;contains hardware I/O equates

;               Miscellaneous storage
CodeHeap        DS.L        8                       ;heap header for kernel heap item
StackEnd        DS.L        40                      ;system stack before tasking starts
StackBegin      DS.L        0                       ;top of startup stack area

;               Pre-tasking initialization
;               this code works in single-task mode
;               prior to the invocation of the context switcher
Start                       ;Initial entry.  Calling operating system is still
                            ;alive and kicking at this point.
TakeOver        LEA         ReEntry,A1              ;point to re-entry instruction
                MOVE.L      A1,$20.W                ;move short absolute to the vector
                                                    ;for privilege exceptions
                MOVE        USP,A0                  ;try a privileged instruction.  If it
                            ;works, then we're in priv. mode. If not, then trap to
                            ;ReEntry and be in privileged mode anyway
ReEntry         LEA         StackBegin,A7           ;set up initial stack

;               Turn off all interrupts in the system
;               Note: this is device-specific code.
;               The labels in the operand fields are from our own
;               SysIO include file.
                CLR.B       FDCIntMask              ;clear floppy disk & system console
                CLR.B       HDIntMask               ;clear hard disk completion int. mask
                CLR.B       SerIO1IntMask           ;clear serial boards
                CLR.B       SerIO2IntMask

;               Initialize the vector table
;               Copy the vectors from an assembled table (in another module)
;               into the actual hardware vector list in low RAM
                LEA         VecTable,A0             ;source (in another code segment)
                LEA         $0.W,A1                 ;destination (begins at $00 0000)
                MOVE        #191,D7                 ;192 longwords to move
VecMove         MOVE.L      (A0)+,(A1)+             ;move a longword
                DBRA        D7,VecMove              ;repeat till done (fast loop on 68010)

;               Copy system routine JMP table from assembled object code (in
;               another module) to low memory jump table, where everyone
;               can get at them.
                LEA         JMPTable,A0             ;source
                LEA         System.W,A1             ;dest. (name of first system call in
                            ;the jump table.  "System" is from the SysEqu include
                            ;file.  It's the context switcher)
                MOVE        #JMPTabLen/4,D7 ;number of longwords to move
JPTMove         MOVE.L      (A0)+,(A1)+             ;move a longword
                DBRA        D7,JPTMove              ;repeat till done (fast loop on 68010)

;               Clear low memory to zero (between jmp table and kernel)
                LEA         StackEnd,A1             ;point to top of destination
                            ;and bottom of destination (end of the jump table)
                LEA         System+JMPTabLen.W,A0
                SUBA        A0,A1                   ;calculate the length
                MOVE.L      A1,D7                   ;move to D7 for counting
                LSR.L       #4,D7                   ;divide by 16 for 16-byte blocks
LowClr          CLR.L       (A0)+                   ;clear 16 bytes, quickly
                CLR.L       (A0)+
                CLR.L       (A0)+
                CLR.L       (A0)+
                DBRA        D7,LowClr               ;do it until done.
```

```
;               Clear high memory to zero (between kernel and end of RAM)
;               (RAMEnd is first byte beyond RAM, defined in SysEqu)
                LEA         RAMEnd,A1               ;point to top of destination
                            ;and bottom of destination (end of the jump table)
                LEA         KernEnd,A0
                SUBA        A0,A1                   ;calc the length
                MOVE.L      A1,D7                   ;move to D7 for counting
                LSR.L       #4,D7                   ;divide by 16 for 16-byte blocks
HiClr           CLR.L       (A0)+                   ;clear 16 bytes, quickly
                CLR.L       (A0)+
                CLR.L       (A0)+
                CLR.L       (A0)+
                DBRA        D7,HiClr                ;do it until done.

;               Initialize all of the primary I/O devices
;               Note: this is a device specific routine not treated in the article.
                JSR         IOInit

;               Initialize the heap
;               Note: this is a routine in the heap manager, which creates
;               valid heap headers for the three initial heap items discussed
;               in the text: the deletion below the kernel, the kernel code
;               item, and the deletion above the kernel.
                JSR         HeapInit

;               Initialize the system zone of low memory
;               Note: this sets up the TCB and master handle arrays, as
;               discussed in the text, as well as initializing the time of day and
;               the date and the other miscellaneous system values.
                JSR         SysInit

;               Spawn off the initial tasks
;               This will create TCBs and TData items for the tasks, but won't
;               invoke them.  They're invoked only by the context switcher.
                LEA         SysConMon,A0            ;point to system console entry point
                MOVE.L      #4096,D0                ;tell it how much RAM for TData
                JSR         Spawn                   ;jump through jump table entry
                            ;("Spawn" is a jump table equate in SysEqu)
                LEA         HeapMunger,A0 ;spawn the heap munger
                MOVE.L      #512,D0                 ;heap munger's TData size
                JSR         Spawn
                LEA         DiskMunger,A0           ;Spawn the disk munger
                MOVE.L      #8192,D0                ;(TData includes one disk buffer)
                JSR         Spawn

                LEA         TCB1.W,A2               ;get address of first TCB in array
                                                    ;(TCB1 is defined in SysEqu)
                BRA.S       ConSw1                  ;now start the context switcher!

;               Context Switcher: primary version
;               Simple task-switch, nothing fancy.
;               SysFlags is a low-RAM system flag byte, defined in SysEqu.
;               The data structure for the TData item is defined in SysEqu.
;               The data structure for the TCB is defined in SysEqu.
ConSwitch       BTST        #StopSys,SysFlags.W ;task switching inhibited?
                BNE.S       ConSwX                  ;yes, exit back to caller
                MOVE.L      OurTCB(A5),A0           ;get TCB address from TData
                SUBA.L      A5,SP                   ;subtract TData base addr from stack
                MOVE.L      SP,TCBSP(A0)            ;save relative displacement in TCB

                MOVE.L      TCBNxt(A0),A2           ;get address of next TCB
ConSw1          MOVE.L      TCBA5(A2),A5            ;get new TData base address
                MOVE.L      TCBSP(A2),SP            ;get stack relative displacement
                ADDA.L      A5,SP                   ;restore absolute address
ConSwX          RTS                                 ;return to next task

;               Context Switcher: alternate version
;               Put the calling task to sleep.
ConSwSleep      BTST        #StopSys,SysFlags.W ;task switching inhibited?
                BNE.S       ConSwX                  ;yes, exit back to caller
                                                    ;without going to sleep
                MOVE.L      OurTCB(A5),A0           ;get TCB address from TData
                SUBA.L      A5,SP                   ;subtract TData base addr from stack
                MOVE.L      SP,TCBSP(A0)            ;save relative displacement in TCB

                MOVE.L      TCBNxt(A0),A2           ;get address of next TCB
                MOVE.L      TCBPrev(A0),A1 ;get addr of previous TCB
                MOVE.L      A2,TCBNxt(A1)           ;close the pointers around the now-
                MOVE.L      A1,TCBPrev(A2) ;sleeping task.
                MOVE.B      #Sleep,TCBState(A0) ;mark it as asleep

                MOVE.L      TCBA5(A2),A5            ;get new TData base address
                MOVE.L      TCBSP(A2),SP            ;get stack relative displacement
                ADDA.L      A5,SP                   ;restore absolute address
                RTS                                 ;return to next task
```

End Listing

LISTING ONE (Text begins on page 76)

```
*************************************************************************
*                                                                       *
*                                                                       *
*       8080 Simulator for MC68000                                      *
*                                                                       *
*       With CP/M 2.2 call support, optional tracing and                *
*       Morrow HDDMA DMA buffer translating.                            *
*                                                                       *
*                                                                       *
*       Version 1.2 1/21/85 JEC                                         *
*               Fixed Extent bug in OPEN logic.                         *
*               Sped up code, sample MAC from 2:13 to 1:40.             *
*               Now runs at a 1.4 MHz equivalent based on MAC sample.   *
*                                                                       *
*       Version 1.1 8/29/84 JEC                                         *
*               Fixed BDOS call #6 bug.                                 *
*                                                                       *
*       Version 1.0 05/25/84 by Jim Cathey                              *
*                                                                       *
*       This program has been written for speed wherever possible,      *
*       as such tends to be large because of the separate subroutine    *
*       for each and every opcode of the target processor.              *
*                                                                       *
*       On an 8MHz 68000 (Compupro) system the simulation speed is      *
*       a little better than a 1MHz Z-80 when running MAC.  The time    *
*       for a sample assembly was 2:13 for the simulation vs 0:35       *
*       on a 4MHz Z-80, both systems used identical hard disk systems.  *
*                                                                       *
*       It is not a complete simulation, as some flag handling          *
*       isn't quite right, but it is enough to run the programs         *
*       I wrote it for (DDT, LU, MAC, and Morrow's FORMATMW).           *
*                                                                       *
*************************************************************************
        text
        page
*************************************************************************
*                                                                       *
*       This file contains the startup routines, the simulator core,    *
*       tracing code, and the CP/M 2.2 simulation.                      *
*                                                                       *
*************************************************************************


        xdef optabl,flags,mnops
        globl mloop,illegl,service

*
*       Conditional assembly flags.
*
trace   equ 0           ; Non-zero for trace routine inclusion.
trcdsk  equ 0           ; Non-zero for FCB trace routine inclusion.
dmpdsk  equ 0           ; Non-zero for register dump in FCB trace.
*  !! diskio is in file COM2.S !!
*diskio equ 0           ; Non-zero for special HDDMA support.


*
*       Register definitions for the simulation.
*
return    equ @16,r     ; JMP (return) is fast return to MLOOP.
pseudopc equ @15,r      ; 8080's PC is register A5.
opptr     equ @14,r     ; Pointer to opcode dispatch table.
pseudosp equ @13,r      ; 8080's SP is register A3.
flagptr  equ @12,r      ; Pointer to 8080's flag lookup table is A2.
targbase equ @11,r      ; Pointer to 8080's address space is A1.
regs      equ @11,r     ; Base pointer to 8080's registers is A1.
regcon0e equ 7,r        ; Register based constant #$E (for speed).
regcon01 equ 6,r        ; Register based constant #$1.
regcon0f equ 5,r        ; Register based constant #$F.
regconff equ 4,r        ; Register based constant #$FF.
regf      equ 3,r       ; 8080's Flags
rega      equ 2,r       ; 8080's Accumulator

*
*       Note, only leaves D0-D1/A0 for free use by entire
*       program without saving registers for temporary use.
*

bdos    .opd 0,$4e42    ; BDOS 'macro'.
bios    .opd 0,$4e43    ; BIOS 'macro'.

        page
*************************************************************************
*                                                                       *
*       Initialization and Main Opcode dispatcher.                      *
*                                                                       *
*************************************************************************

start   lea.l target,targbase   ; Start of target memory.
        ifne trace              ; Optional trace code.
        bsr entrads             ; Enter trace delimiting addresses
*                               ; if the code is desired.
        endc
        bsr lodfdos             ; Load up the fake FDOS in target mem.
        bsr lodregs             ; Load the remaining simulation registers.
        bsr loadcom             ; Load the .COM program,
        tst d0                  ; quit if unsuccessful.
        bne optprnt
        rts

optprnt equ *
        ifne trcdsk             ; If FCB tracing, print header.
        lea.l fcbmsg,a0
        bsr lpstr
        endc


mloop:  *                       ; Execute simulation
~~mloop:
        ifne trace              ; Optional trace.
        tst traceflg
        bne dotrace
        cmpa.l tracesad,pseudopc
        bne notrace
        move.b #1,traceflg
dotrace bsr dump
        cmpa.l traceead,pseudopc
        bne notrace
        move.b #0,traceflg
notrace equ *
        endc

        moveq #0,d0             ; Execute appropriate simulation subroutine
        move.b (pseudopc)+,d0   ; Grab next opcode.
        asl #2,d0               ; (D0 high word is still 0!)
        move.l 0(opptr,d0.w),a0
        jmp (a0)                ; To the subroutine.

        page
*************************************************************************
*                                                                       *
*       Illegal instructions and Dumping.                               *
*                                                                       *
*************************************************************************

illegl  move.l #illgmsg,d1      ; Illegal opcode, say what & where,
        move.w #9,d0
        bdos
        lea.l -1(pseudopc),a0
        move.b (a0),d1
        suba.l targbase,a0
        bsr pbyte
        move.l #ilgmsg2,d1
        move.w #9,d0
        bdos
        move.l a0,d1
        bsr pword
        move.l #ilgmsg3,d1
        move.w #9,d0
        bdos
        move.l #dumpmsg,d1
        move.w #9,d0
        bdos
        bsr dump                ; and spill guts.
        rts                     ; Quit simulation.

        page
dump    movem.l d0-d1/a0,-(sp)
        move.l #dmpmsg2,d1      ; Dump all registers,
        move.w #9,d0            ; used for illegals and tracing.
        bdos
        move.b rega,d1
        bsr pbyte
        move.b regf,d1
        bsr pbyte
        bsr pspace
        move.w regb(regs),d1
        bsr pword
        bsr pspace
        move.w regd(regs),d1
        bsr pword
        bsr pspace
        move.w regh(regs),d1
        bsr pword
        bsr pspace
        move.l pseudosp,d1
        sub.l targbase,d1
        bsr pword
        bsr pspace
        move.l pseudosp,a0
        swap d2                 ; Save REGA
        move.w #3,d2
tosloop move.b 1(a0),d1
        ror.w #8,d1
        move.b 0(a0),d1
        bsr pword
        bsr pspace
        addq.l #2,a0
        dbra d2,tosloop
        swap d2
        move.l pseudopc,d1
        sub.l targbase,d1
        bsr pword
```

```
         bsr pspace
         bsr pspace
         move.b (pseudopc),d1
         bsr pbyte
         bsr pspace                 ; Now show mnemonic
         bsr pspace
         moveq #0,d0
         move.b (pseudopc),d0
         asl.w #2,d0
         lea.l mnops,a0
         move.l (a0,d0.l),d1
         move.l d1,-(sp)
         inc.l d1
         move #9,d0
         bdos
         move.l (sp)+,a0
         cmp.b #" ",(a0)
         beq nooprnd
         cmp.b #"C",(a0)
         bne notcons
         move.b 1(pseudopc),d1
         bsr pbyte
         bra nooprnd
notcons  cmp.b #"A",(a0)
         bne nooprnd
         move.b 2(pseudopc),d1
         bsr pbyte
         move.b 1(pseudopc),d1
         bsr pbyte
nooprnd  bsr pspace                 ; In case of conout calls during trace,
         bsr pspace                 ; they will be visible at end of line.
         bsr pspace
         movem.l (sp)+,d0-d1/a0
         rts

         page
*************************************************************************
*                                                                       *
*        Initialization subroutines.                                    *
*                                                                       *
*************************************************************************

lodfdos lea.l fdos,a6           ; Load up the fake FDOS.
         move.l targbase,pseudosp
         adda.l #$10000,pseudosp
         lea.l -256(pseudosp),a0
         move.w #fdoslen,d0
lodloop  move.b (a6)+,(a0)+
         dbra d0,lodloop
         lea.l -256(pseudosp),a0
         move.l a0,d0
         sub.l targbase,d0
         move.b #$c3,0(targbase) ; Build BIOS & BDOS jumps.
         move.b #$c3,5(targbase)
         move.b d0,6(targbase)
         rol.w #8,d0
         move.b d0,7(targbase)
         rol.w #8,d0
         add.w #3,d0
         move.b d0,1(targbase)
         rol.w #8,d0
         move.b d0,2(targbase)
         move.w #0,-(pseudosp)   ; Set up a return stack to exit simulation.
         rts


lodregs  lea.l optabl,opptr      ; Point base register to opcode dispatch
                                 table.
         lea.l mloop,return
         lea.l flags,flagptr
         move.l targbase,pseudopc
         adda.l #$100,pseudopc   ; Start execution at 0100H in target space.
         moveq #$e,regcon0e      ; Set up quick constants.
         moveq #$1,regcon01
         moveq #$f,regcon0f
         move.l #$ff,regconff
         moveq #0,rega
         moveq #0,regf
         rts

         page
entrads  move.l #tracemsg,d1     ; Enter trace address if necessary.
         move.w #9,d0
         bdos
```

(Continued on next page)

LISTING ONE **(Listing Continued, text begins on page 76)**

```
        bsr atol                ; Get trace start address.
        and.l #$ffff,d1
        move.l d1,a0
        adda.l targbase,a0
        move.l a0,tracesad
        move.l #tracemg2,d1
        move.w #9,d0
        bdos
        bsr atol                ; Get trace end address.
        and.l #$ffff,d1
        move.l d1,a0
        adda.l targbase,a0
        move.l a0,traceead
        move.w #10,d1           ; CRLF to end line.
        move.w #2,d0
        bdos
        move.w #13,d1
        move.w #2,d0
        bdos
        rts

*
*       OPEN file to be loaded, and load it into target
*       space if successful.
*
loadcom link a6,#0              ; Mark stack frame.
        movem.l d2-d3/a2-a4,-(sp)
        move.l 12(a6),a0        ; Get the address of the base page.
        lea.l $5c(a0),a2        ; Get FCB address.
        move.b #'C',9(a2)       ; mash filename to .COM
        move.b #'O',10(a2)
        move.b #'M',11(a2)
        move.l a2,d1
        move.w #15,d0
        bdos                    ; OPEN file.
        cmpi.w #255,d0          ; ERROR?
        beq openerr

        move.l pseudopc,d2      ; Start loading at $0100 in target.
filelod move.l d2,d1            ; Set DMA address.
        move.w #26,d0
        bdos
        move.l a2,d1
        move.w #20,d0           ; Read file until EOF.
        bdos
        tst d0
        bne basepg
        add.l #128,d2
        bra filelod

basepg  lea.l $80(targbase),a2  ; Set up the target's base page.
        move.l a2,d1            ; Start with default DMA address.
        move.l a2,dmaaddr
        move.w #26,d0
        bdos
        lea.l $38(a0),a2
        lea.l $5c(targbase),a3  ; Copy host's 2nd FCB to target's 1st FCB.
        move.w #36,d0
fcbloop move.b (a2)+,(a3)+
        dbra d0,fcbloop
        lea.l $80(a0),a2
        lea.l $80(targbase),a4
        lea.l $81(targbase),a3
        clr d0
        move.b d0,(a4)
        move.b (a2)+,d0         ; Grab command tail from host's buffer.
tail1   cmp.b #$20,(a2)+        ; Hack off ?.COM filename.
        dbeq d0,tail1
        bne loaded              ; If there's any tail left, then
tail2   cmp.b #$20,(a2)+        ; remove leading whitespace.
        dbne d0,tail2
        beq loaded
        dec.l a2
        subq #2,d0
tail3   move.b (a2)+,(a3)+      ; Move the rest of the tail.
        inc.b (a4)
        dbra d0,tail3
        move.b #0,(a3)
        bra loaded

openerr move.l #opnmsg,d1       ; Can't open file.
        move.w #9,d0
        bdos
        clr d0

loaded  movem.l (sp)+,d2-d3/a2-a4
        unlk    a6              ; Trantor.
        rts


        page
***********************************************************************
*                                                                     *
*       BIOS and BDOS service request handler.                        *
*                                                                     *
***********************************************************************
```

```
service moveq #0,d0                     ; Handle BIOS/BDOS service request
        move.b (pseudopc)+,d0           ; of form HLT DB opcode.
        bne biosfn                      ; BDOS or BIOS?
bdosfn  moveq #0,d1
        move.b regc(regs),d0            ; Get BDOS function number.
        move.w regd(regs),d1            ; Get argument.
        cmp #31,d0                      ; Can't do Disk Parm Hdr function
        beq badbdos
        cmp #27,d0                      ; or ALLOC vector fn.
        bne okbdos
badbdos move.l #ilgbdos,d1
        move.w #9,d0
        bdos
        bsr dump
        bra quitprg

okbdos  cmp #9,d0                       ; Translate target address to real address.
        blt noconv
        cmp #14,d0
        beq noconv
        cmp #32,d0
        beq noconv
        cmp #37,d0
        beq noconv
        add.l targbase,d1
noconv  cmp #26,d0                      ; Save last known DMA address
        bne notdma                      ; (in case of OPEN processing).
        move.l d1,dmaaddr
notdma  move.b #0,fcbflag               ; Separate FCB type requests
        cmp #15,d0                      ; from the rest of the swine.
        blt notfcb                      ; (Assume not, at first).
        cmp #24,d0
        blt fcb
        cmp #30,d0
        beq fcb
        cmp #33,d0
        blt notfcb
        cmp #37,d0
        blt fcb
        cmp #40,d0
        beq fcb
        bra notfcb


        page
fcb     swap d2
        move.w #35,d2                   ; Move the FCB to host working buf,
        move.l d1,a0
        move.l a1,-(sp)
        lea.l fcbstor,a1
fcb1    move.b (a0)+,(a1)+
        dbra d2,fcb1
        move.l (sp)+,a1
        lea.l fcbstor,a0                ; and swap the random record bytes
        move.b 33(a0),d2                ; to make them match the 68000's.
        move.b 35(a0),33(a0)
        move.b d2,35(a0)
        swap d2
        move.b #1,fcbflag               ; Set flag for proper recovery.
        move.l d1,-(sp)                 ; (Gotta put the pig back in pen!)
        move.l a0,d1
        ifne trcdsk
        ifne dmpdsk                     ; Optional^2 Register dump.
        bsr dump
        endc
        endc
        cmp.w #15,d0                    ; OPEN has a problem in that CP/M-68K
        bne notopen                     ; can only open the base extent, unlike
        tst.b 12(a0)                    ; CP/M-80.  So we have to check and do
        beq notopen                     ; an OPEN then SEEK (RREAD) if required.
        bsr openproc
        bra results

notopen:
~~notopen:
        ifne trcdsk                     ; Optional FCB trace.
        move.l d2,-(sp)
        move.b #' ',d2
        bsr fcbtrc1
        move.l (sp)+,d2
        endc


notfcb  cmp #6,d0                       ; Not an FCB request.  Is it
        bne notdcon                     ; a direct console I/O function?
        cmp.b #$ff,d1                   ; Yes, make host's look like target's.
        bne notdcon
        move.w #$fe,d1
        bdos
        tst d0
        beq results
        move.w #6,d0
        move.w #$FF,d1
notdcon bdos                            ; FINALLY!  Do the translated function.
results move.w d0,regh(regs)
        move.b d0,rega
```

```
        move.b regh(regs),regb(regs)
        tst.b fcbflag           ; Do we need to restore a FCB?
        beq done
        ifne trcdsk
        bsr fcbtrc2
        endc
        lea.l fcbstor,a0        ; Restore the FCB to target, in proper
                                  order.
        swap d2
        move.b 33(a0),d2
        move.b 35(a0),33(a0)
        move.b d2,35(a0)
        move.l (sp)+,a0
        move.l a1,-(sp)
        lea.l fcbstor,a1
        move.w #35,d2
fcb2    move.b (a1)+,(a0)+
        dbra d2,fcb2
        swap d2
        move.l (sp)+,a1
done    move.b rega,d0
        and.w regconff,d0
        move.b 0(flagptr,d0.w),regf
        rts


openproc:
~~openproc:
        ifne trcdsk             ; Optional FCB trace.
        swap d2
        move.b #' ',d2
        bsr fcbtrc1
        swap d2
        bsr fcbtrc2a
        endc

        move.b 33(a0),-(sp)     ; Save away RR fields!
        move.b 34(a0),-(sp)
        move.b 35(a0),-(sp)
        movem.l d0-d2,-(sp)
        moveq #0,d2
        move.b 12(a0),d2        ; Save desired extent.
        clr.b 12(a0)
        bsr fcbbdos             ; Do BDOS (with opt. tracing).
        tst.b d0
        bmi badopen             ; No seek if not good OPEN.
        asl.l #7,d2             ; Make EXTENT # into record offset.
        moveq #0,d0
        move.b 32(a0),d0
        bclr #7,d0
        add.l d2,d0             ; Add onto CR to make abs record #.
        move.w d0,34(a0)        ; Put into FCB.
        swap d0
        move.b d0,33(a0)
        move.l #junkbuf,d1      ; Set DMA addr elsewhere for Rand Seek.
        move.w #26,d0
        bdos
        movem.l (sp)+,d0-d2
        move.w #33,d0           ; Random READ (SEEK) desired extent.
        bsr fcbbdos             ; Do BDOS (with opt. tracing).
        clr d0                  ; (OPEN) must always be successful because
*                               ; of the way CP/M-80 & CP/M-68K differ
*                               ; on OPENing non-zero extents.
        movem.l d0-d1,-(sp)     ; Restore the proper DMA address.
        move.w #26,d0
        move.l dmaaddr,d1
        bdos
        movem.l (sp)+,d0-d1
restore move.b (sp)+,35(a0)     ; Restore RR fields.
        move.b (sp)+,34(a0)
        move.b (sp)+,33(a0)
        rts

badopen movem.l (sp)+,d0-d2
        bra restore


fcbbdos:
~~fcbbdos:
        ifne trcdsk             ; BDOS call with optional FCB trace.
        move.l d2,-(sp)
        move.b #'+',d2
        bsr fcbtrc1
```

(Continued on next page)

LISTING ONE (Listing Continued, text begins on page 76)

```
        move.l (sp)+,d2
        endc
        bdos
        ifne trcdsk
        bsr fcbtrc2
        endc
        rts


biosfn  cmp #1,d0               ; Handle Bios calls.
        beq quitprg
        cmp #$f,d0              ; List Status is ok.
        beq gudbios
        cmp #7,d0
        bge badbios             ; Don't allow disk functions!
gudbios clr.w d1
        move.b regc(regs),d1
        movem.l d2-d7/a0-a6,-(sp)
        bios
        movem.l (sp)+,d2-d7/a0-a6
        move.b d0,rega
        rts

badbios move.b d0,-(sp)         ; Flag illegal BIOS call
        move.l #biosmsg,d1      ; and spill guts.
        move.w #9,d0
        bdos
        move.b (sp)+,d1
        bsr pbyte
        move.l #biosmg2,d1
        move.w #9,d0
        bdos
        bsr dump

quitprg move.l (sp)+,d0         ; Trash return address and
        rts                     ; quit simulation.

        page
****************************************************************************
*                                                                          *
*       FCB Tracing support routines.                                      *
*                                                                          *
****************************************************************************

        ifne trcdsk
fcbtrc1 movem.l d0-d2/a0,-(sp)  ; Dump to printer each FCB usage
        move.b #9,d1            ; in format FN #, Disk, Name (ASCII)
        bsr lpchar              ; and the rest, all in hex but the
        move.w d0,d1            ; name field.  Print the returned
        bsr lpbyte              ; value after the FCB.
        move.b d2,d1            ; Char in D2 is printed after FN #.
        bsr lpchar
        bsr lpspace
        bsr lpspace
        move.b (a0)+,d1
        bsr lpbyte
        bsr lpspace
        move.w #10,d2
fcbtr1  move.b (a0)+,d1         ; Print Name field...
        bsr lpchar
        dbra d2,fcbtr1
        bsr lpspace
        move.w #3,d2
fcbtr2  move.b (a0)+,d1         ; Ex .. Rc
        bsr lpbyte
        bsr lpspace
        dbra d2,fcbtr2
        bsr lpspace
        bsr lpspace
        lea.l 16(a0),a0         ; Skip d0..dn field.
        move.w #3,d2
fcbtr3  move.b (a0)+,d1         ; CR .. R2
        bsr lpbyte
        bsr lpspace
        dbra d2,fcbtr3
        bsr lpspace
        bsr lpspace
        move.l dmaaddr,d1
        sub.l targbase,d1
        bsr lpword
        bsr lpspace
        movem.l (sp)+,d0-d2/a0
        rts

        page
fcbtrc2 movem.l d0-d1,-(sp)     ; Line termination of FCB trace.
        bsr lpspace
        bsr lpspace
        move.b d0,d1
        bsr lpbyte
fcbtr21 move.b #10,d1
        bsr lpchar
        move.b #13,d1
        bsr lpchar
        movem.l (sp)+,d0-d1
        rts
fcbtrc2a:
        movem.l d0-d1,-(sp)     ; Line termination if no result
        bra fcbtr21             ; is to be presented.
        endc

        page
****************************************************************************
*                                                                          *
*       Misc. service routines.                                            *
*       (Inelegant, but rarely used so they stand as is).                  *
*                                                                          *
****************************************************************************

pbyte   move.l #$20018,d0       ; 2 nybbles, 24 bit shift first.
        bra pdigits
pword   move.l #$40010,d0       ; 4 nybbles, 16 bit shift first.
        bra pdigits
paddr   move.l #$60008,d0       ; 6 nybbles, 8 bit shift first.
        bra pdigits
plong   move.l #$80000,d0       ; 8 nybbles, no shift first.
pdigits rol.l d0,d1             ; Do shift.
        bra pdigent
pdiglop swap d0                 ; Save nybble count.
        rol.l #4,d1             ; Print variable in d1.
        bsr ntoa
pdigent swap d0                 ; Get nybble count.
        dbra d0,pdiglop
        rts

ntoa    movem.l d0-d1,-(sp)     ; Nybble in d1 to ASCII, then output.
        and #$f,d1
        cmp #$a,d1
        blt ntoa2
        add.b #'A'-'9'-1,d1
ntoa2   add.b #'0',d1
        move.w #2,d0
        bdos
        movem.l (sp)+,d0-d1
        rts

pspace  move.w #32,d1           ; Print a space.
        move.w #2,d0
        bdos
        rts

        page
*
*       Line Printer versions of above
*

lpbyte  move.l #$20018,d0       ; 2 nybbles, 24 bit shift first.
        bra lpdigts
lpword  move.l #$40010,d0       ; 4 nybbles, 16 bit shift first.
        bra lpdigts
lpaddr  move.l #$60008,d0       ; 6 nybbles, 8 bit shift first.
        bra lpdigts
lplong  move.l #$80000,d0       ; 8 nybbles, no shift first.
lpdigts rol.l d0,d1             ; Do shift.
        bra lpdgent
lpdiglp swap d0                 ; Save nybble count.
        rol.l #4,d1             ; Print variable in d1.
        bsr lntoa
lpdgent swap d0                 ; Get nybble count.
        dbra d0,lpdiglp
        rts

lntoa   movem.l d0-d1,-(sp)     ; Nybble in d1 to ASCII, then output.
        and #$f,d1
        cmp #$a,d1
        blt lntoa2
        add.b #'A'-'9'-1,d1
lntoa2  add.b #'0',d1
lntoa3  move.w #5,d0
        bdos
        movem.l (sp)+,d0-d1
        rts


lpchar  movem.l d0-d1,-(sp)     ; Print a character.
        bra lntoa3

lpspace movem.l d0-d1,-(sp)     ; Print space.
        move.w #32,d1
        bra lntoa3

        page
*
*       Remaining misc. service routines.
*

lpstr   movem.l d0-d1,-(sp)     ; Print a null-terminated string.
lpstr1  move.b (a0)+,d1
        beq lpstr2
        bsr lpchar
        bra lpstr1
```

```
lpstr2  movem.l (sp)+,d0-d1
        rts


konin   move.w #1,d0     ; Console input for 'atol'.
        bdos
        rts


atol    moveq #0,d1      ; ASCII to long, stops on invalid hex char.
        clr d2           ; Returns long in d1, terminator char in d0,
atol1   bsr konin        ; d2=1 if any chars entered before terminator.
        cmp.b #$40,d0
        blo atol2
        and #$5F,d0      ; Mask to upper case.
atol2   cmpi.b #'0',d0   ; Check range (0..9,A..F).
        blo atolend
        cmpi.b #'F',d0
        bhi atolend
        cmpi.b #'9',d0
        bls atol3
        cmpi.b #'A',d0
        bhs atol3
        bra atolend
atol3   moveq #1,d2      ; Valid characters entered, flag it.
        sub.b #'0',d0
        cmp.b #$9,d0
        bls atol4
        sub.b #'A'-'9'-1,d0
atol4   ext d0           ; To long.
        ext.l d0
        asl.l #4,d1      ; Tack it onto D1.
        add.l d0,d1
        bra atol1
atolend rts

        page
        data
*********************************************************************
*                                                                   *
*       Target processor's data registers.                          *
*       Fake FDOS.                                                  *
*                                                                   *
*********************************************************************

        even
regop3  equ -9           ; Operand 1 for DAA storage.
regb    equ -8           ; Offsets from register base pointer for
regc    equ -7           ; 8080's pseudo-registers.
regd    equ -6           ; A & F are in Data Registers.
rege    equ -5           ; Pseudo-PC is kept in an Address Register.
regh    equ -4
regl    equ -3
regop1  equ -2           ; Operand 1 for DAA storage.
regop2  equ -1           ;    "     2  "    "      "

fcbstor ds.b 36          ; Host works FCB's out of here.
fcbflag ds.b 1           ; Flag says we used the FCB buffer.

        even
tracesad ds.l 1          ; Trace start address.
traceead ds.l 1          ; Trace end address.
traceflg ds.w 1          ; Tracing enabled flag.

dmaaddr ds.l 1           ; DMA address storage.

        page
fdos    dc.b $76,0,$C9   ; Fake BDOS for target system.
*                        ; BIOS Jump Table
        dc.b $C3,$33,$FF        ; Wboot
        dc.b $C3,$36,$FF        ; Const
        dc.b $C3,$39,$FF        ; Conin
        dc.b $C3,$3C,$FF        ; Conout
        dc.b $C3,$3F,$FF        ; List
        dc.b $C3,$42,$FF        ; Punch
        dc.b $C3,$45,$FF        ; Reader
        dc.b $C3,$48,$FF        ; Home
        dc.b $C3,$4B,$FF        ; Seldsk
        dc.b $C3,$4E,$FF        ; Settrk
        dc.b $C3,$51,$FF        ; Setsec
        dc.b $C3,$54,$FF        ; Setdma
        dc.b $C3,$57,$FF        ; Read
        dc.b $C3,$5A,$FF        ; Write
        dc.b $C3,$5D,$FF        ; Listst
        dc.b $C3,$60,$FF        ; Sectran
```

(Continued on next page)

LISTING ONE (Listing Continued, text begins on page 76)

```
        dc.b $76,1,$C9  ; Fake BIOS for target system
        dc.b $76,2,$C9          ; Const
        dc.b $76,3,$C9          ; Conin
        dc.b $76,4,$C9          ; Conout
        dc.b $76,5,$C9          ; List
        dc.b $76,6,$C9          ; Punch
        dc.b $76,7,$C9          ; Reader
        dc.b $76,8,$C9          ; Home *
        dc.b $76,9,$C9          ; Seldsk *
        dc.b $76,1Ø,$C9         ; Settrk *
        dc.b $76,11,$C9         ; Setsec *
        dc.b $76,12,$C9         ; Setdma *
        dc.b $76,13,$C9         ; Read *
        dc.b $76,14,$C9         ; Write *
        dc.b $76,15,$C9         ; Listst
        dc.b $76,16,$C9         ; Sectran *

fdoslen equ *-fdos

        page
*******************************************************************
*                                                                 *
*       Messages.                                                 *
*                                                                 *
*******************************************************************

illgmsg dc.b $d,$a,'Illegal instruction $'
ilgmsg2 dc.b ' at $'
ilgmsg3 dc.b '.$'
dumpmsg dc.b $d,$a,'Register contents:$'
dmpmsg2 dc.b $d,$a
        dc.b '-AF- -BC- -DE- -HL- -SP- -SØ- -S1- -S2- -S3- -PC- -op-',$d,
         $a,'$'
biosmsg dc.b $d,$a,'Illegal BIOS call $'
biosmg2 dc.b '.$'
tracemsg dc.b 13,1Ø,'Start trace at >$'
tracemg2 dc.b 13,1Ø,'End trace at >$'
opnmsg  dc.b 'Cannot open .COM file.$'
ilgbdos dc.b 'Unsupported BDOS call.$'
fcbmsg  dc.b 9,'Fn# Dr    NAME     EX S1 S2 RC   CR RØ R1 R2   Addr  Rslt'
         ,1Ø,13
        dc.b 9,'-----------------------------------------------------------'
         ,1Ø,13,Ø
        page
        bss

*******************************************************************
*                                                                 *
*       Target processor's address space.                         *
*                                                                 *
*******************************************************************

        even
registers ds.b 1Ø               ; Actual storage for 8Ø8Ø's other registers.
target  ds.b $1ØØØØ             ; 8Ø8Ø's universe.
junkbuf ds.b $8Ø                ; For BDOS' OPEN faking (RREAD buffer).
        .end
```

End Listing One

LISTING TWO

```
*******************************************************************
*                                                                 *
*       This file contains the target processor (8Ø8Ø) simulation *
*       routines.                                                 *
*                                                                 *
*******************************************************************

*******************************************************************
*                                                                 *
*       Opcode dispatch table.  One longword entry per opcode of the *
*       target (8Ø8Ø) processor, including illegals.             *
*                                                                 *
*******************************************************************


        globl optabl,flags,nopØØ
        xdef mloop,illegl,service,preED,outspec

diskio   equ Ø          ; Non-zero for special HDC/DMA support.

return   equ @16,r      ; JMP (return) is fast return to MLOOP.
pseudopc equ @15,r      ; 8Ø8Ø's PC is register A5.
opptr    equ @14,r      ; Pointer to opcode dispatch table.
pseudosp equ @13,r      ; 8Ø8Ø's SP is register A3.
flagptr  equ @12,r      ; Pointer to 8Ø8Ø's flag lookup table is A2.
targbase equ @11,r      ; Pointer to 8Ø8Ø's address space is A1.
regs     equ @11,r      ; Base pointer to 8Ø8Ø's registers is A1.

regconØe equ 7,r        ; Register based constant #$E (for speed).
regconØ1 equ 6,r        ; Register based constant #$1.
regconØf equ 5,r        ; Register based constant #$F.
regconff equ 4,r        ; Register based constant #$FF.
*                       ;  Often used constants #Ø & #8 are predominantly
*                       ;  used by instructions that have 'quick' modes
*                       ;  which encompass these values -- no register
*                       ;  needed (or available, either).
regf     equ 3,r        ; 8Ø8Ø's Flags
rega     equ 2,r        ; 8Ø8Ø's Accumulator

regop3   equ -9         ; Operand 3 for DAA storage.
regb     equ -8         ; Offsets from register base pointer for
regc     equ -7         ; 8Ø8Ø's pseudo-registers.
regd     equ -6         ; A & F are in Data Registers.
rege     equ -5         ; Pseudo-PC is kept in an Address Register.
regh     equ -4
regl     equ -3
regop1   equ -2         ; Operand 1 for DAA storage.
regop2   equ -1         ;    "    2  "   "    "

        data
        page
        even
optabl  dc.l nopØØ,lxib,staxb,inxb,inrb,dcrb,mvib,rlca
        dc.l nopØ8,dadb,ldaxb,dcxb,inrc,dcrc,mvic,rrca
        dc.l nop1Ø,lxid,staxd,inxd,inrd,dcrd,mvid,ral
        dc.l nop18,dadd,ldaxd,dcxd,inre,dcre,mvie,rar
        dc.l nop2Ø,lxih,shld,inxh,inrh,dcrh,mvih,daa
        dc.l nop28,dadh,lhld,dcxh,inrl,dcrl,mvil,cma
        dc.l nop3Ø,lxis,sta,inxs,inrm,dcrm,mvim,stc
        dc.l nop38,dads,lda,dcxs,inra,dcra,mvia,cmc
        dc.l movebb,movebc,movebd,movebe,movebh,movebl,movebm,moveba
        dc.l movecb,movecc,movecd,movece,movech,movecl,movecm,moveca
        dc.l movedb,movedc,movedd,movede,movedh,movedl,movedm,moveda
        dc.l moveeb,moveec,moveed,moveee,moveeh,moveel,moveem,moveea
        dc.l movehb,movehc,movehd,movehe,movehh,movehl,movehm,moveha
        dc.l movelb,movelc,moveld,movele,movelh,movell,movelm,movela
        dc.l movemb,movemc,movemd,moveme,movemh,moveml,halt,movema
        dc.l moveab,moveac,movead,moveae,moveah,moveal,moveam,moveaa
        dc.l addb,addc,addd,adde,addh,addl,addm,addaa
        dc.l adcb,adcc,adcd,adce,adch,adcl,adcm,adca
        dc.l subb,subc,subd,sube,subh,subl,subm,subaa
        dc.l sbbb,sbbc,sbbd,sbbe,sbbh,sbbl,sbbm,sbba
        dc.l andb,andc,andd,ande,andh,andl,andm,anda
        dc.l xrab,xrac,xrad,xrae,xrah,xral,xram,xraa
        dc.l orab,orac,orad,orae,orah,oral,oram,oraa
        dc.l cmpb,cmpc,cmpd,cmpe,cmph,cmpl,cmpam,cmpaa
        dc.l rnz,popb,jnz,jmpa,cnz,pushb,adi,rstØ
        dc.l rz,ret,jz,nopCB,cz,call,aci,rst8
        dc.l rnc,popd,jnc,out,cnc,pushd,sui,rst1Ø
        dc.l rc,nopD9,jc,in,cc,nopDD,sbi,rst18
        dc.l rpo,poph,jpo,xthl,cpo,pushh,ani,rst2Ø
        dc.l rpe,pchl,jpe,xchg,cpe,preED,xri,rst28
        dc.l rp,popp,jp,di,cp,pushp,oria,rst3Ø
        dc.l rm,sphl,jm,ei,cm,nopFD,cpi,rst38

        page
*******************************************************************
*                                                                 *
*       Flag register lookup tables.                              *
*                                                                 *
*******************************************************************

flags dc.b $ØØ,$Ø1,$Ø4,$Ø5,$4Ø,$41,$44,$45,$8Ø,$81,$84,$85,$CØ,$C1,$C4,$C5

      dc.b $44,$ØØ,$ØØ,$Ø4,$ØØ,$Ø4,$Ø4,$ØØ,$ØØ,$Ø4,$Ø4,$ØØ,$Ø4,$ØØ,$ØØ,$Ø4
      dc.b $ØØ,$Ø4,$Ø4,$ØØ,$Ø4,$ØØ,$ØØ,$Ø4,$Ø4,$ØØ,$ØØ,$Ø4,$ØØ,$Ø4,$Ø4,$ØØ
      dc.b $ØØ,$Ø4,$Ø4,$ØØ,$Ø4,$ØØ,$ØØ,$Ø4,$Ø4,$ØØ,$ØØ,$Ø4,$ØØ,$Ø4,$Ø4,$ØØ
      dc.b $Ø4,$ØØ,$ØØ,$Ø4,$ØØ,$Ø4,$Ø4,$ØØ,$ØØ,$Ø4,$Ø4,$ØØ,$Ø4,$ØØ,$ØØ,$Ø4
      dc.b $ØØ,$Ø4,$Ø4,$ØØ,$Ø4,$ØØ,$ØØ,$Ø4,$Ø4,$ØØ,$ØØ,$Ø4,$ØØ,$Ø4,$Ø4,$ØØ
      dc.b $Ø4,$ØØ,$ØØ,$Ø4,$ØØ,$Ø4,$Ø4,$ØØ,$ØØ,$Ø4,$Ø4,$ØØ,$Ø4,$ØØ,$ØØ,$Ø4
      dc.b $Ø4,$ØØ,$ØØ,$Ø4,$ØØ,$Ø4,$Ø4,$ØØ,$ØØ,$Ø4,$Ø4,$ØØ,$Ø4,$ØØ,$ØØ,$Ø4
      dc.b $ØØ,$Ø4,$Ø4,$ØØ,$Ø4,$ØØ,$ØØ,$Ø4,$Ø4,$ØØ,$ØØ,$Ø4,$ØØ,$Ø4,$Ø4,$ØØ
      dc.b $8Ø,$84,$84,$8Ø,$84,$8Ø,$8Ø,$84,$84,$8Ø,$8Ø,$84,$8Ø,$84,$84,$8Ø
      dc.b $84,$8Ø,$8Ø,$84,$8Ø,$84,$84,$8Ø,$8Ø,$84,$84,$8Ø,$84,$8Ø,$8Ø,$84
      dc.b $84,$8Ø,$8Ø,$84,$8Ø,$84,$84,$8Ø,$8Ø,$84,$84,$8Ø,$84,$8Ø,$8Ø,$84
      dc.b $8Ø,$84,$84,$8Ø,$84,$8Ø,$8Ø,$84,$84,$8Ø,$8Ø,$84,$8Ø,$84,$84,$8Ø
      dc.b $84,$8Ø,$8Ø,$84,$8Ø,$84,$84,$8Ø,$8Ø,$84,$84,$8Ø,$84,$8Ø,$8Ø,$84
      dc.b $8Ø,$84,$84,$8Ø,$84,$8Ø,$8Ø,$84,$84,$8Ø,$8Ø,$84,$8Ø,$84,$84,$8Ø
      dc.b $8Ø,$84,$84,$8Ø,$84,$8Ø,$8Ø,$84,$84,$8Ø,$8Ø,$84,$8Ø,$84,$84,$8Ø
      dc.b $84,$8Ø,$8Ø,$84,$8Ø,$84,$84,$8Ø,$8Ø,$84,$84,$8Ø,$84,$8Ø,$8Ø,$84

        page
*******************************************************************
*                                                                 *
*       Opcode simulation subroutines.                            *
*                                                                 *
*       Note:   I/O instructions are based at 68ØØØ address $FFØØØØ *
*               as is appropriate for the CompuPro CPU-68K card.  *
*                                                                 *
*       Also, all routines assume that the high word of DØ = Ø!   *
*                                                                 *
*******************************************************************

        text
        even

nopØØ   jmp (return)                            ; ØØ Nop
```

```
lxib    move.b (pseudopc)+,regc(regs)       ; 01 Lxi BC,nnnn
        move.b (pseudopc)+,regb(regs)
        jmp (return)

staxb   move.w regb(regs),d0                ; 02 Stax B
        move.b rega,0(targbase,d0.l)
        jmp (return)

inxb    inc.w regb(regs)                    ; 03 Inx B
        jmp (return)

inrb    inc.b regb(regs)                    ; 04 Inr B
        move sr,d0
        and.w regcon0e,d0
        and.w regcon01,regf
        or.b 0(flagptr,d0.w),regf
        jmp (return)

dcrb    dec.b regb(regs)                    ; 05 Dcr B
        move sr,d0
        and.w regcon0e,d0
        and.w regcon01,regf
        or.b 0(flagptr,d0.w),regf
        jmp (return)

mvib    move.b (pseudopc)+,regb(regs)       ; 06 Mvi b,nn
        jmp (return)

rlca    rol.b #1,rega                       ; 07 Rlc
docyf   bcs rlc1
        bclr #0,regf
        jmp (return)
rlc1    bset #0,regf
        jmp (return)

nop08   bra illegl                          ; 08 Illegal for 8080

dadb    move.w regb(regs),d0                ; 09 Dad B
        add.w d0,regh(regs)
        bra docyf

ldaxb   move.w regb(regs),d0                ; 0A Ldax B
        move.b 0(targbase,d0.l),rega
        jmp (return)

dcxb    dec.w regb(regs)                    ; 0B Dcx B
        jmp (return)

inrc    inc.b regc(regs)                    ; 0C Inr C
        move sr,d0
        and.w regcon0e,d0
        and.w regcon01,regf
        or.b 0(flagptr,d0.w),regf
        jmp (return)

dcrc    dec.b regc(regs)                    ; 0D Dcr C
        move sr,d0
        and.w regcon0e,d0
        and.w regcon01,regf
        or.b 0(flagptr,d0.w),regf
        jmp (return)

mvic    move.b (pseudopc)+,regc(regs)       ; 0E Mvi C
        jmp (return)

rrca    ror.b #1,rega                       ; 0F Rrc
        bra docyf

nop10   bra illegl                          ; 10 Illegal for 8080

lxid    move.b (pseudopc)+,rege(regs)       ; 11 Lxi DE,nnnn
        move.b (pseudopc)+,regd(regs)
        jmp (return)

staxd   move.w regd(regs),d0                ; 12 Stax D
        move.b rega,0(targbase,d0.l)
        jmp (return)

inxd    inc.w regd(regs)                    ; 13 Inx D
        jmp (return)
```

(Continued on next page)

LISTING TWO (Listing Continued, text begins on page 76)

```
inrd    inc.b regd(regs)                        ; 14 Inr D
        move sr,dØ
        and.w regconØe,dØ
        and.w regconØ1,regf
        or.b Ø(flagptr,dØ.w),regf
        jmp (return)

dcrd    dec.b regd(regs)                        ; 15 Dcr D
        move sr,dØ
        and.w regconØe,dØ
        and.w regconØ1,regf
        or.b Ø(flagptr,dØ.w),regf
        jmp (return)

mvid    move.b (pseudopc)+,regd(regs)           ; 16 Mvi D,nn
        jmp (return)

ral     roxr.b #1,regf                          ; 17 Ral
        roxl.b #1,rega
        roxl.b #1,regf
        jmp (return)

nop18   bra illegl                              ; 18 Illegal for 8Ø8Ø

dadd    move.w regd(regs),dØ                    ; 19 Dad D
        add.w dØ,regh(regs)
        bra docyf

ldaxd   move.w regd(regs),dØ                    ; 1A Ldax D
        move.b Ø(targbase,dØ.l),rega
        jmp (return)

dcxd    dec.w regd(regs)                        ; 1B Dcx D
        jmp (return)

inre    inc.b rege(regs)                        ; 1C Inr E
        move sr,dØ
        and.w regconØe,dØ
        and.w regconØ1,regf
        or.b Ø(flagptr,dØ.w),regf
        jmp (return)

dcre    dec.b rege(regs)                        ; 1D Dcr E
        move sr,dØ
        and.w regconØe,dØ
        and.w regconØ1,regf
        or.b Ø(flagptr,dØ.w),regf
        jmp (return)

mvie    move.b (pseudopc)+,rege(regs)           ; 1E Mvi E,nn
        jmp (return)

rar     roxr.b #1,regf                          ; 1F Rar
        roxr.b #1,rega
        roxl.b #1,regf
        jmp (return)

nop2Ø   bra illegl                              ; 2Ø Illegal for 8Ø8Ø

lxih    move.b (pseudopc)+,regl(regs)           ; 21 Lxi H,nnnn
        move.b (pseudopc)+,regh(regs)
        jmp (return)

shld    move.b 1(pseudopc),dØ                   ; 22 Shld addr
        rol.w #8,dØ
        move.b (pseudopc),dØ
        addq.l #2,pseudopc
        move.l dØ,aØ
        adda.l targbase,aØ
        move.b regl(regs),(aØ)+
        move.b regh(regs),(aØ)
        jmp (return)

inxh    inc.w regh(regs)                        ; 23 Inx H
        jmp (return)

inrh    inc.b regh(regs)                        ; 24 Inr H
        move sr,dØ
        and.w regconØe,dØ
        and.w regconØ1,regf
        or.b Ø(flagptr,dØ.w),regf
        jmp (return)

dcrh    dec.b regh(regs)                        ; 25 Dcr H
        move sr,dØ
        and.w regconØe,dØ
        and.w regconØ1,regf
        or.b Ø(flagptr,dØ.w),regf
        jmp (return)

mvih    move.b (pseudopc)+,regh(regs)           ; 26 Mvi H,nn
        jmp (return)

daa     move.b regop3(regs),dØ                  ; 27 Daa
        roxr.b dØ
        move.b regop2(regs),dØ
        move.b regop1(regs),d1
        swap regconØe
        move.b rega,regconØe
        and.b regconØf,regconØe
        cmp.b #9,regconØe
        bhi halfcy
        and.b regconØf,dØ
        and.b regconØf,d1
        ori.b #$fØ,d1
        addx.b dØ,d1
        bcc nohalf
halfcy  add.b #6,rega
        bcs fullcy
nohalf  btst #Ø,regf
        bnz fullcy
        move.b rega,regconØe
        and.b #$fØ,regconØe
        cmp.b #$9Ø,regconØe
        bls nofull
```

(Listings to be Continued next month)

ANNOUNCING! DR. DOBB'S COMPLETE TOOLBOX OF C

Dr. Dobb's Journal, the most respected source of technical software information available, brings you this collection of powerful programming tools for C.

New, from M&T Publishing and Brady Communications . . .

Dr. Dobb's Toolbook for C

Item #005

A comprehensive library of valuable C code.

Many of Dr. Dobb's most popular articles on C from sold-out issues are updated and reprinted in this unique reference, along with new C programming tools. **THE TOOLBOX** contain's C-compilers, line editors, assemblers, text processing programs, and more! Dr. Dobb's Journal offers **The Toolbook** in a special hardbound edition for only $29.95. You'll find:

- J.E. Hendrix's famous *Small C Compiler v. 2* and *A New Library for Small C* (also available on disk)
- Never before published: Hendrix's new *Small-Mac: An Assembler for Small C* and *Small Tools: Programs for Text Processing* (both available on disk)
- Ron Cain's original *A Small-C Compiler for the 8080's*
- Plus many useful programming tools in C

Also from M&T Publishing and Brady Communication—

The Small-C Handbook

Item #006 or #006A

The Small-C Handbook by James Hendrix is a valuable resource of information about the Small-C Compiler. In addition to descriptions of the language and the compiler, **The Handbook** also contains the entire source listings of the compiler and its library of arithmetic and logical routines. A perfect companion to the Hendrix Small-C compiler offered by Dr. Dobb's on disk, **The Handbook** even tells how to use the compiler to generate new versions of itself. All this is in **The Handbook** for only $17.95, Item #006; only $22.95 for **The Handbook** with the **MS/PC DOS Handbook Addendum**, Item #006A.

While both **The Toolbook** and **The Handbook** provide documentation for the Small-C Compiler on disk, **The Handbook** provides more complete documentation and is available with an Addendum for MS/PC DOS versions.

LISTING ONE (Text begins on page 118)

```
;
; DUMMY SEGMENT TO BE GROUPED WITH HIMEM, WHICH
; IS AN EMPTY SEGMENT AT THE END OF THE USER'S
; PROGRAM
;
HIMEM GROUP HIDATA
HIDATA SEGMENT COMMON 'HIMEM'
FLARB DW 0
HIDATA ENDS
;
;
; 'MAIN' IS THE PROGRAM'S ENTRY POINT,
; JUST ABOVE THE PSP
;
EXTRN MAIN:FAR
;
ARCODE SEGMENT PUBLIC 'CODE'
;
      ASSUME CS:ARCODE
      PUBLIC FREMEM
;
FREMEM PROC FAR
;
      PUSH BP                  ; STANDARD ENTRY STUFF
      MOV  BP,SP               ;
;
; GET THE ADDRESS OF HIMEM, THE TOP OF THE
; USER'S PROGRAM, STICK IT IN USER'S VARIABLE
;
      MOV  AX,SEG HIDATA    ; GET TOP SEGMENT
      LES  BX,18[BP]        ; SEND IT TO THE USER
      MOV  ES:[BX],AX       ;
;
;
; FIND OUT HOW MUCH MEMORY IS AVAILABLE ABOVE
; HIMEM BY REQUESTING A LOT OF MEMORY.
; THE CALL TO FUNCTION 4AH NEEDS THE SEGMENT
; ADDRESS OF THE PROGRAM'S PSP. GET IT BY
; SUBTRACTING 10 PARAGRAPHS FROM THE SEGMENT
; OF THE PROGRAM'S ENTRY POINT, WHICH HAS THE
; LABEL 'MAIN'.
;
      MOV  AX,SEG MAIN      ; GET ENTRY SEGMENT
      SUB  AX,10H           ; ADJUST TO GET PSP
      MOV  ES,AX            ; PUT VALUE INTO ES
      MOV  BX,-1D           ; GET ALL OF MEMORY
      MOV  AX,4A00H         ; DOS FUNCTION CODE
      INT  21H              ; DO IT
;
; THE NUMBER OF AVAILABLE PARAGRAPHS IS IN BX,
; RETURN IT TO USER.
;
      LES  DI,14[BP]        ; STORE IT IN
      MOV  ES:[DI],BX       ; USER'S VARIABLE
;
;
; TRY TO ALLOCATE THE PARAGRAPHS REQUESTED.
; A VALUE OF -1 MEANS DON'T ALLOCATE ANY.
;
      LES  DI,10[BP]        ; GET THE USER'S
      MOV  BX, ES:[DI]      ; VARIABLE
      CMP  BX,-1D           ; SHOULD WE ALLOCATE?
      JE   OK               ; NO, BAIL OUT
      MOV  AX,SEG MAIN      ; GET ENTRY POINT
      SUB  AX,10H           ; ADJUST TO GET PSP
      MOV  ES,AX            ; PUT IT IN ES
      MOV  AX,4A00H         ; DOS FUNCTION CODE
      INT  21H              ; DO IT
      JNA  ERR              ; CHECK FOR ERRORS
OK:   XOR  AX,AX            ; NONE, CLEAR FLAG
ERR:  LES  BX,6[BP]         ; STORE ERROR CODE
      MOV  ES:[BX],AX       ; IN USER'S VARIABLE
;
; THAT'S IT
;
      MOV  SP,BP            ; STANDARD EXIT STUFF
      POP  BP
      RET  16
;
FREMEM ENDP
;
ARCODE ENDS
       END
```

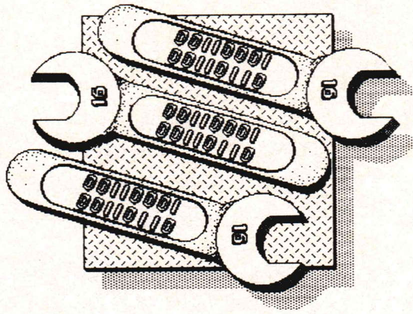
End Listing One

LISTING TWO

```
; Michael Barr's 32-bit Square Root Routine
;
; Call with CX:BX = argument
; Returns   BX    = result
;
sqrt    proc    near        ;CX:BX = argument
        push    ax          ;save other registers
        push    dx
        push    di
        jcxz    sqrt3
        mov     dx,cx       ;prepare first try
        mov     di,-1
sqrt1:  shl     dx,1        ;estimating size of arg.
        jc      sqrt2       ;to guess initial try
        shl     dx,1
        jc      sqrt2
        shr     di,1
        jmp     sqrt1
sqrt2:  mov     dx,cx       ;restore argument
        mov     ax,bx
        cmp     dx,di       ;prevent overflow
        jae     sqrt4
        div     di
        cmp     ax,di       ;comp quotient and divisor
        jae     sqrt4
        add     di,ax       ;average them
        rcr     di,1
        jmp     sqrt2       ;and do it again
sqrt3:  mov     dx,bx       ;prepare first try
        mov     di,0ffh
        or      bx,bx       ;lower half zero?
        jnz     sqrt1       ;no, jump
        mov     di,bx
sqrt4:  mov     bx,di       ;return result in BX
        pop     di
        pop     dx
        pop     ax
        ret
sqrt    endp
```

End Listings

Trojan Horse Programs



One of the more ominous developments in the last few months is the appearance of so-called Trojan horse programs on some public bulletin board systems. These have appeared under such names as EGABTR, VDIR, and SYSUTIL. They are usually accompanied by scanty documentation that bills them as super disk directories or something similar, but their actual effect is to trash your system by formatting the hard disk, erasing the file allocation table, or writing random garbage into files. I put the people who create and upload such programs to public BBSs in the same category as terrorists.

Along similar lines, some villains have taken advantage of their knowledge of the public-domain PC remote BBS to upload programs that purport to paint a pretty picture on the screen or do something else cute but actually copy the password file or other vital system files to (unprotected) files under another name. The villain calls back later and downloads the unprotected files, thereby gaining access to privileged files and messages.

As the computer terrorists become more clever, we can expect the appearance of subtle sabotage programs that copy themselves to hidden files on the hard disk or attach themselves to the bootable operating system and don't do their damage until much later. For example, I can envision a hard disk destruction program that would wait until it saw that you had not run Backup for a week! The nature of the damage could also be so subtle that it would drive you crazy, such as simply changing a bit in a random sector of the hard disk every few days.

These developments have the potential to damage or destroy the proliferation of public-domain software on bulletin boards, which would be very sad. BBS operators will not want to take the risk of being held liable for disasters due to Trojan horse programs. On the LMI RBBS, we are immediately adopting a policy of deleting all programs that are not uploaded in the form of, or at least accompanied by, source code as a clear-text ASCII file. I'd like to hear comments from readers on this subject, especially those who have been victims or who can provide actual samples of these Trojan horse programs.

EXEC Calls and FORTRAN

Robert Sypek of a firm called Argis in Hudson, Massachusetts, writes: "While attempting to write a program that would execute a user or system task from a user program, I ran into many of the problems described in past issues about using the *DOS 2.x EXEC* function call. My problems were compounded by the fact that Microsoft's languages (FORTRAN, Pascal, C) set up their own data and stack segments, and the code segment is that of the user program or subroutine, not the segment of the root program. This means that the program segment prefix (PSP) is inaccessible from any routine called by a user program. All is not lost, however, as I believe I have found a method of retrieving the necessary information to allow an EXEC function to be called.

"The method depends on two pieces of information that may be gleaned from the Microsoft user manuals: the compiler defines a null segment called *HIMEM* that is placed at the end of all other segments when the user program is loaded, and the compiler defines a symbol called *MAIN* that is located directly after the PSP. Both the segment and the symbol have the *PUBLIC* attribute, making them accessible to the user.

"The segment of the PSP may be found by subtracting 10H from the segment of *MAIN*, yielding the value of the ES register needed for the call to *EXEC*. The value of *HIMEM* can then be used to calculate the size of the user program and the segment of the next available paragraph of memory."

Mr. Sypek enclosed an assembly-language subroutine that we are printing as Listing One (page 116). The routine is invoked in the form:

```
CALL FREMEM(MEMPTR,MEMAVL,
    MEMALL,ERR)
```

where *MEMPTR* is the segment value of the next available paragraph, *MEMAVL* is the number of paragraphs available, and *MEMALL* is a user-defined variable specifying the amount of memory above the program to leave allocated to the user. (A value of -1 leaves all memory allocated to the program.) An error code of 0 indicates that the function was successful.

Lightweight Reading

Those who think the mainframe mentality is gone forever should read Martin Healy's article "Toward a Viable OS for the PC" (*Datamation*, September 1985). At first I took this article for a practical joke because it contains so many distortions of the history and current state of the art in microcomputers, sideways slams at a variety of targets, and outrageous gobbledygook (example: "PC Network could be the answer, but it won't share data, only the file").

by Ray Duncan

The programs published in this month's column are available for downloading from the Laboratory Microsystems RBBS at (213) 306-3530 (300 baud or 1,200 bps).

The author asserts that "there are in fact two leading real operating systems for micros, Unix and Concurrent DOS from Digital Research." He goes on to say, "The new Version 4.1 [of Concurrent DOS] is the idealized multitasking PC DOS, which due to its maturity should eliminate any Microsoft version (PC DOS 4.0?) thereof. That leaves Microsoft to concentrate on its Unix-like system, Xenix." I am sure the folks at Microsoft will be very relieved to learn from Mr. Healy that they no longer have to waste all that effort on maintaining MS DOS, can take MS DOS 4.0 out of testing and toss it in the trash can, and turn their attention to other matters.

Square Roots

Michael Barr of the Department of Mathematics at McGill University sent us his 8086 assembly-language subroutine for square roots (Listing Two, page 116). He writes: "This routine gives the correct floored square root for any 32-bit number (considered as unsigned). It is also faster than the bit-at-a-time algorithms you have put into *DDJ*.

"Apropos that last statement, there seems to be a discussion between people who believe that Newton's method is always the best way to do a square root and others who believe equally fervently that the bit-at-a-time method is always faster. Common sense would dictate that they are both wrong. I strongly suspect that Newton's method is faster if and only if you have an on-chip (or coprocessor) division of the relevant size. In particular, to do a 32-bit square root, you need a 32-bit by 16-bit division. This much I have tested; Newton's method is just about twice as fast (about 330 msec, compared to about 650 msec for the bit-at-a-time). What I haven't tested (I can't face the thought of programming them) are 64-bit square roots. But there is every reason to believe that it will be faster to do the bit-at-a-time square root than to code division and then use that for Newton's method."

Big DOS

The new wave of PCs, based on the 80286 microprocessor, are still limited to 1 megabyte of direct memory addressing because they run MS DOS or its clones in 8086 emulation mode (called Real Mode by Intel). The ability of the 80286 to address 16 megabytes of RAM in "Protected Virtual Memory Mode" is currently supported only by the 80286 Xenix and iRMX-286 operating systems, neither of which is likely to become very popular due to their incompatibility with WordStar, dBASE III, and Lotus 1-2-3. The average user's exploitation of the full capabilities of the PC/AT and other such machines awaits the arrival of an operating system that runs in Protected Mode and can execute the popular MS DOS applications unaltered.

Such an operating system has been the subject of much industry rumor and speculation. Digital Research has been talking about "Concurrent DOS-286" for some time now, even announcing the operating system's "immediate availability" at a press conference in New York about six months ago. (See "Concurrent DOS-286: Available Now," *Intel Speak Softly Quarterly*, vol. 2, no. 2, Second Quarter 1985.) But lately DRI has been backpedaling a bit and now admits

(Continued from page 119)

that 80286 Concurrent DOS will probably never exist in the form originally advertised. Of course, the company is blaming its problems on "defects" in the 80286 design. (When in doubt, fall back on the classic programmer's defense: "It must be a hardware problem.")

Microsoft Corp. is also known to be working on a Protected Mode, upward-compatible version of PC DOS (already dubbed Big DOS by the trade rags and assumed to be Version 5.0). But Microsoft has been very close-mouthed about these efforts, presumably wanting to make sure it can pull it off before committing itself to such a product in public. Too bad DRI wasn't that careful!

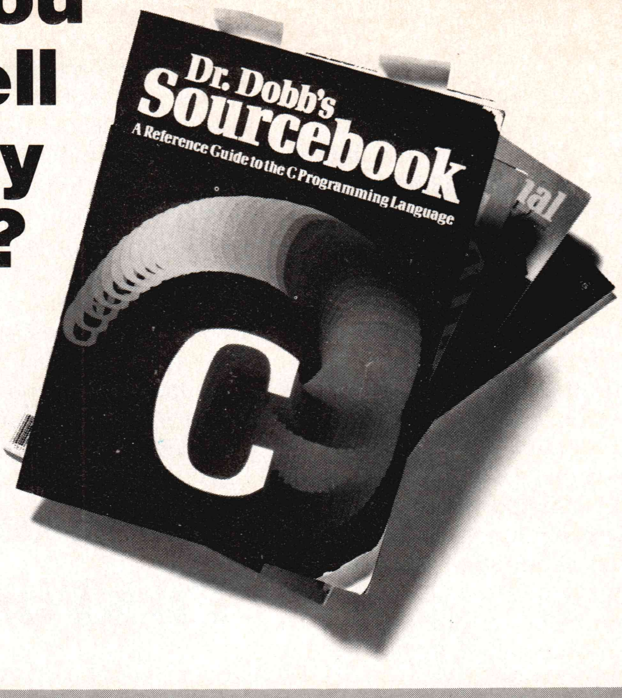
Ross Nelson, who previously worked on the 80286 team at Intel, took the time to write to us with some musings on the future of DOS and Protected Mode. He says, "With regard to the 286 in the marketplace, it seems to me that unless a comparatively low-cost system (PC/II?) is introduced, there won't be a lot of work done that will take advantage of Protected Mode (PM). The installed base of AT users vs. the number of PC-compatible users will make it economically unfeasible. Once people do start working with PM, they will encounter some interesting problems. The great virtue of PM is that no task in the system should be able to corrupt another task (assuming the operating system is stable). As a software engineer, I applaud this philosophy, and I believe that this use of the 286 should be encouraged. Realistically, however, it is clear that there will be a transition period in which PM will only be used to gain access to the larger memory space.

Nelson wrote: "As far as I can tell, there are only two ways of switching back to Real Mode when you are in Protected Mode, and only one of them is feasible with the standard 80286 part. This is essentially the method IBM has chosen [in the VDISK driver supplied with PC DOS 3.x . . . RD], which is to place enough state information to restart your process in a 'safe' location and RESET the processor. The other method requires a special 'bond-out' part (which Intel uses in its I2-ICE). By activating a special pin on the bond-out chip, you can issue special instructions that dump and restore the internal state of the machine, including the Machine Status Word. Systems built with the bond-out chip could easily be toggled between Real Mode and Protected Mode.

"Whether or not a DOS 5.0 or Big DOS can be successful in emulating the current PC system architecture on a 286 will depend on how freely the implementors translate the word *compatible*. I do not believe that 100 percent compatibility can be accomplished without unreasonable overhead. I suspect that even partial compatibility will have large memory requirements. It would not surprise me to see a 512K operating system with an additional 32K required on a per-task basis. Here is how I believe some of the problems that come up can be handled:

"Video—This has quite a few simple solutions: (a) trap each initial write to the display segment, and replace the offending segment register with

Who Says You Can't Tell A Book By Its Cover?



Dr. Dobb's Sourcebook: A Reference Guide for the C Programming Language

For years, serious programmers have relied on Dr. Dobb's Journal for the technical tools of their trade. Now, Dr. Dobb's presents the definitive programmers guide to the who, what, where, when and why of C, the leading language among software developers. This comprehensive guide to new information, products and services specific to C will be your most often-used reference!

In this valuable guide you'll find:

- An extensive directory of hardware and software services—including classes and seminars, C programming opportunities, and on-line services
- A bibliography with over 300 listings of available articles and books on C
- A comprehensive C product listing—including C compilers, graphics modules, utilities, editors and development systems, and more!
- And much more practical C programming information

At only $7.95, no C programmer can afford to be without this unique reference.

To order by credit card, call toll free: 1-800-528-6050 ext.4001 . Ask for item 004 or mail this coupon, along with payment to **Dr. Dobb's Journal, 2464 Embarcadero Way, Palo Alto, CA 94303**

PAYMENT MUST ACCOMPANY YOUR ORDER

_____ I enclose check/money order

_____ Please charge my ___VISA ___M/C ___ American Express

Card #________________________ Exp. Date _____

Signature ________________________

Name ________________________

Address ________________________

City ______________ State ______ Zip ______

Please allow 6 to 12 weeks for delivery

Please send me _______ copies of **Dr. Dobb's Sourcebook**

at $7.95 each = ____________

\+ Shipping & Handling = ____________

(Must be included with order. Please add $1.50 per book in U.S. $3.25 each surface mail outside U.S. Foreign airmail rates available on request.)

TOTAL = ____________

3111D

(Continued from page 121)

an OS-created descriptor; (b) use the TopView solution, that is, require applications to issue a software interrupt to get the address of their own 'local' display buffer; (c) trap every display write and deal with it on a case-by-case basis. Solution (c) is the most compatible but the poorest performer; (a) is almost as compatible and would substantially increase performance. Solution (b) is the best and would not cause a great deal of incompatibility, especially with programs that use installable device drivers.

"Communications—Here we have no simple solutions. I would expect that all but the most primitive communications applications would have to be rewritten. Anything more complicated than Int 14H calls should be declared incompatible and rewritten to fit the new OS.

"EXE files—Here, the OS must limit the 'free-memory' size to 64K of code and 64K of data. Programs that need more memory should be forced to [dynamically] allocate it. Because of the large number of programs that indiscriminately load segment registers, however, an OS might want to trap segment load faults and attempt to map them into the 8086-style physical addressing scheme.

"COM files—These programs are the most likely to be incompatible, because they often load and stay resident and have only one segment (CS). Because executable segments are never writable, an OS would incur severe performance penalties trying to simulate the standard DOS mode of operation here. Some heuristics would work for some programs, such as loading DS, SS, and ES with a writable alias of the CS descriptor, but any program that did segment register arithmetic would then yield incorrect results.

"Obviously, there are hundreds of similar problems. . . . The question is, what solutions will the marketplace accept? It seems that a Protected Mode 286 operating system will have to contain a large measure of DOS/IBM PC compatibility to prevent the 'software gap' problem faced by the Macintosh, Amiga, etc. Unfortunately, this means a lower-performance, aesthetically unpleasing solution. I would welcome a radically different, optimized system, but only IBM has the clout to pull it off, and there would still be two important factors: IBM would have to want to do it, and it would have to do it right."

Nothing but the Best for My Little Girl

Richard Gaulden, vice president of sales and marketing for EDCOM, Inc., wrote me a letter describing his company as a "national supplier of enrichment materials to school and library systems." His sales pitch wound up with the statements: "EDCOM can free your distribution channels by marketing software that becomes outdated by new releases. The educational market does not have the need for constant upgrades, especially when new versions are more costly." Kind of gives you a warm, secure feeling about your kids' education in computer literacy, doesn't it?

DDJ

(Listing begins on page 116)

Reader Ballot
Vote for your favorite feature/article.
Circle Reader Service **No. 6.**

Kitchen-Table Entrepreneur

The purpose of this department is to provide information of use to professional programmers. This first installment addresses itself to the independent software developer, romantically and perhaps not altogether inaccurately envisioned as working on a VAX by day and by night designing the compiler of tomorrow on an XT on the kitchen table. We point here to a number of books that might be of use to the kitchen-table entrepreneur.

Copyright

Read books on copyright with some caution. Both copyright law itself and the interpretations of it in court have changed in their application to software publishing. Don't assume that you need not copyright your software if you favor a Freeware/ Shareware approach to distribution.

Remer, Daniel. *Legal Care for Your Software*. Berkeley, Calif.: Nolo Press, 1984.

Read this for discussion of all the legal issues: work-for-hire agreements, nondisclosure agreements, contracts, license agreements. The author covers the differences among and nuances of copyright, trade-secret protection, patent, and trademark registration. The book explains your legal liabilities and how international copyright works. It has a number of sample forms, such as a beta test site agreement form.

Salone, M.J. *How to Copyright Software*. Berkeley, Calif.: Nolo Press, 1984.

Read it for deeper discussion of copyright issues. What happens when the first 500 disks go out with no copyright notice on the disk or in the code? What can you do if your copyright is infringed? Hundreds of examples are provided.

Documentation

Read something on documentation and realize how important it is. A good manual is, after all, the best copy protection available. Many programmers cut their own throats by releasing good software with illiterate, poorly designed documentation. Besides creating an aura of amateurishness for your product, such a manual will hide the capabilities of your product from users—and however experienced a programmer you may be, the odds are you're an amateur in producing documentation.

Houghton-Alico, Doann. *Creating Computer Software User Guides*. New York: McGraw-Hill, 1985.

Read it to jog your memory about some of the techniques available for communicating about your software—maybe what your compiler documentation needs is a good pie chart. Maybe not. Use the book to get an idea of what professional documentation writers have to do, though their tasks are somewhat different from yours, of course.

Stephan, Peter M. *Writing User-usable Manuals*. Salt Lake City: Wredco Press, 1984.

Read it as an example of decent low-cost documentation. The author has won awards for his documentation, and although the book may not tell you anything you don't already know, it shows that a manual need not be typeset and perfect bound.

Strunk, William, and White, E.B., *The Elements of Style*. New York: Macmillan, 1972.

Read it. Whether you write the documentation or hire writers, your words are likely to see print somewhere. There is no more concise guide to keeping your foot out of your semiliterate mouth than this book.

Markets

Read these if you're really an independent software developer and don't want to distribute your product(s) yourself. These books mainly list software distributors.

Amato, Francis. *Guide to Computer Magazines*. Dallas: Steve Davis, 1985.

Read it for a quick idea of the editorial focus, circulation, and audience of selected computer magazines. This can be useful for promotional and advertising purposes, and some of the magazines are software markets in themselves, publishing programs in their pages and/ or on disks.

Hoffman, Roger. *The Complete Software Marketplace*. New York: Warner Books, 1984.

Read it for many reasons. One reason is to get an overview of the things you need to know in running a small software business. The book has lists of distributors, including mail-order companies, electronic distributors, and magazines. There are also lists of disk duplication services, PR firms, market research firms, trade associations and shows, seminars, and conferences. There are case studies and a useful section on freebies for the independent author.

McGehee, Brad M., ed. *1986 Programmer's Market*. Cincinnati: Writer's Digest Books, 1985.

Read it for publishers who use "free-lance" material—even the terminology reflects this book's heritage. It was put together by people who view the independent software developer as another kind of author. If that description fits you, this book may also.

Software Design

Read this to remind yourself of the basic principles and also to learn a vocabulary for communicating the principles to others.

General Electric. *Software Engineering Handbook*. New York: McGraw-Hill, 1986.

Read it for one approach to the management of large software projects when your design team actually becomes a team. Also read its bibliography for sources on design methods: Nassi/ Sneiderman, Warnier, Yourdon, and Jackson methodologies; data-flow driven design.

DDJ

OF INTEREST

Peak Electronics has announced a 68K coprocessor board that, it claims, you can plug into any (IEEE-696-1983) S-100 system running CP/M-2.2 and have CP/M-68K running in minutes with no hardware or software modifications. The 68K8-CP is a 10-MHz processor card with an MC68008 (8-bit version of the 68000), up to 512K RAM, up to 128K EPROM, two 38.4 kilobaud serial ports, and an 8-bit parallel printer port. The 68K8-CP doesn't replace your current CPU card but runs in conjunction with it, so you can jump back and forth between operating systems. Peak says that the board will support Concurrent DOS 68K when it's released.

Speaking of Concurrent DOS 68K, maybe DRI is finally going to exploit the head start it had in 68K development software: The company was expected to introduce its 68K developer's kit at Comdex in November. With Concurrent, DRI has two operating systems for the 68000; three if you count the GEM windowing environment. Concurrent DOS 68K has a CP/M front end that will run most CP/M-68K programs, and DRI is promising GEM support for subsequent versions of Concurrent DOS 68K.

Speaking of GEM, last month we discussed it and other windowing environments and, because of space limitations, mentioned only some of the most visible. We don't feel right about giving all that attention to the big boys and never mentioning two windowing products we have had good experience with: Desqview and dWindow. It was, of course, not only Digital Research, IBM, and Microsoft that went windowing. Quarterdeck's Desqview is an impressive competitor to TopView, which bundled with AST's Rampage expanded memory board, lets users run up to nine programs concurrently in memory. Also, the success of the concept of windows is exemplified by its employment in a nonoperating system application in the product dWindow. For years, Ashton-Tate's dBASE II set the standard for austerity in user interfaces: You can't get much simpler than a single-period prompt. Liberty Bell Publishing's dWindow does a dazzling cathedral window treatment on dBase II (and now dBase III) that makes it look like an entirely different product.

Speaking of entirely different products, Answer Software has announced an 80286 emulator for developers of 80286-based software. The ICD286 consists of a card for the host system (which must be in the IBM PC/XT/AT family or a compatible), a Buffer Box that plugs into the 80286 slot in the target system, and a symbolic debugger. It allows uploading and downloading of code and data, hardware and software breakpoints, single-stepping, and full-speed emulation up to 10-MHz clock rates.

And speaking of 80286 development, American ADO has introduced an 80286 board for Multibus systems. The SOL C286-01 (no relation to the Processor Technology Sol computer of story and song) is being manufactured in 6-, 8-, and 10-MHz versions and has up to 512K RAM, a Centronics printer interface, and two 8- or 16-bit SBX bus I/O connectors. Then there's the ET-286Plus, a 10-MHz AT-compatible single-board computer that uses the new 1-Mb dynamic RAMs and allows 4 megabytes onboard. ATS International was expected to show it at Comdex.

As long as we're speaking of the 286 and operating systems, we should mention Locus Computing's Multisystem Merge. This product allows simultaneous, transparent execution of Unix and MS DOS on the same machine according to the company. You can set several Unix tasks to work in the background while you run a DOS application in the foreground. This is the system AT&T is using on its 6300+ Unix/DOS computer. Locus developed some of the technology in the system while writing PC-Interface, a product that links DOS computers to a host Unix machine.

DRI, which we were speaking of a few paragraphs back, is of course not the only developer of operating system software for the 68K, as two recent announcements prove. U S Software has announced a real-time multitasking system for embedded applications using the 68K. It's ROMable, requires 3K of code space, and is called USX68K. Integrated Business Computers has ported TheOS-16 (formerly Oasis) to its line of 68010 computers and was expected to be showing a beta version at Comdex.

Speaking of beta-test versions, Tall Tree Systems has begun shipping beta versions of its Jlaser-printer interface to software companies it deems closest to achieving compatible products. The interface is a PC/XT/AT card that works with the JRAM 2-megabyte memory board; it's designed to spend memory to buy print speed and typeface flexibility for laser printers. It transfers bit-mapped images directly from RAM to the print mechanism and is supposed to provide unlimited type fonts with full graphics capabilities at 300 dots per inch in eight seconds.

Speaking of *mucho* megabytes, Reference Technology has announced a device that lets you attach up to eight of its optical disk drives, for more than 4 gigabytes of storage on a (sturdy, large) desktop. The device is PC/XT/AT or compatible compatible.

Speaking of product announcements, Speech Technology is now selling the cross-assemblers it developed in the process of designing and manufacturing electronic devices to aid the blind (its real business). The MS DOS cross-assemblers for the 8048 and 6502 were written in C and support a subset of C preprocessor commands, macros, three object file formats, and features to support PROM programming. They sell for $30 each or $75 for

both with source code and are distributed with no restriction on noncommercial copying.

Speaking of copying, we are watching with interest SoftKlone's fortunes with a product it cheerfully describes as a clone of Microstuf's Crosstalk XVI data-communications package. SoftKlone's Mirror was designed to the precise visual specs of Crosstalk, and SoftKlone presents itself as introducing a new idea—mirror-image software at a lower price than the mirrored product and perhaps with added capabilities. Rather than competing by producing a better or cheaper product, the notion here is to produce the same product better or cheaper.

Reference Map

Peak Electronics, P.O. Box 700112, San Jose, CA 95170; (408) 253-5108. Reader Service Number 16.

Digital Research, P.O. Box DRI, Monterey, CA 93942; (408) 649-3896. Reader Service Number 17.

Liberty Bell Publishing, 618 N.W. Glisan, Ste. 203, Portland, OR 97209; (503) 221-1806. Reader Service Number 18.

Quarterdeck Office Systems, 1918 Main St., Ste. 240, Santa Monica, CA 90405; (213) 392-9851. Reader Service Number 19.

Answer Software, 20863 Stevens Creek Blvd., Cupertino, CA 95014; (408) 253-7515. Reader Service Number 20.

American ADO, 1840 West 186th St., Ste. 200, Torrance, CA 90504; (213) 532-5010. Reader Service Number 21.

ATS International, 2105 Luna Rd., Ste. 330, Carrollton, TX 75006; (214) 247-5151. Reader Service Number 22.

Locus Computing Corp., 3330 Ocean Park Blvd., Santa Monica, CA 90405; (213) 452-2435. Reader Service Number 23.

U S Software, 5470 N.W. Innisbrook Pl., Portland, OR 97229; (503) 645-5043. Reader Service Number 24.

Integrated Business Computers, 21621 Nordhoff St., Chatsworth, CA 91311; (818) 882-9007. Reader Service Number 25.

Tall Tree Systems, 1120 San Antonio Rd., Palo Alto, CA 94303; (415) 964-1980. Reader Service Number 26.

Reference Technology, 1832 North 55th St., Boulder, CO 80301; (303) 449-4157. Reader Service Number 27.

Speech Technology Inc., 16321 176th Ave. N.E., Woodinville, WA 98072; (206) 483-5150. Reader Service Number 28.

SoftKlone, 1210 East Park Ave., Tallahassee, FL 32301; (904) 878-8564. Reader Service Number 29.

DDJ

ADVERTISER INDEX

Reader Service No.	Advertiser	Page No.
158	AMPRO Computers Inc.	125
176	Advent Products Inc.	64-65
221	Alcyon Corp.	52
121	Arity Corporation	117
224	Atari Corporation	C-3
216	Atron	24
159	Blaise Computing	4
161	Borland International	C-4
126	Boston Software Works	101
181	C User's Group	98
209	Certified Software Corp.	97
178	Chalcedony	81
81	Cogitate Inc.	98
122	CompuView	39
225	Computel Publishing Society	116
96	Computer Innovations	72
82	Creative Programming	58
86	Dalsoft Systems	94
130	Data Base Decisions	88
203	Datalight	97
87	Digital Research Computers	53
204	Disclone	83
179	Earth Computers	86
89	Ecosoft Inc.	31
210	Educational Microcomputer Systems	28
90	Edward K. Ream	37
138	Essential Software	77
165	Everest Solutions	C-2
180	Executive Systems	17
211	FOG	75
93	Faircom	56
94	Fox Software Inc.	22
*	Gimpel Software	128
*	Gimpel Software	48
97	Greenleaf Software Inc.	123
98	Guidance Software	107
132	Harvard Softworks	91
134	HiSoft	94
*	House Ad (Back Issues)	80
*	House Ad (Bound Volume)	120
*	House Ad (C Products)	114-115
*	House Ad (Mac Reprint)	113
*	House Ad (Sourcebook)	122
*	House Ad (Subscription)	80
*	House Ad (Z80 Toolbook)	100
166	Inetco Systems	127
194	InfoPro Systems	33
*	Integral Quality	93
215	Kadak Products	36
100	Kriya	125
101	Lattice Inc.	57
135	Lugaru Software, Ltd.	51
218	MacMemory Inc.	105
223	Manx Software	87
222	Manx Software	89
109	Manx Software	96
108	Manx Software	7
102	Mark Williams	11
189	Martian Technologies	109
199	Media Cybernetics	55
220	Metadigm Inc.	90
110	Micro Interfaces Corp.	43
136	Microcomputer Systems Consultants	42
*	Micromint Inc.	84
105	MicroProcessors Unlimited	101
*	Microtec Research	113
190	Mitek	97
*	Mix Software	59
128	Morgan Computing Company	23
79	Mystic Canyon Software	119
137	OES Systems	127
124	Optotech	1
192	Overland Data Inc.	50
200	Pecan Software Systems	13
76	Personal Tex Inc.	50
139	Phoenix Computer Products	46
91	Phoenix Computer Products	5
193	Plu Perfect Systems	24
169	Poor Person Software	105
*	Precise Electronics	111
201	Prentice-Hall Inc.	98
196	Pro/Am Software	54
140	Productivity Products Int'l.	73
143	Programmer's Shop	35
141	Programmer's Shop	63
205	Quelo	94
107	Quilt Computing	101
206	Raima Corp.	19
145	Rational Systems Inc.	92
170	Relational Database Systems	82
213	68 Micros	111
78	SLR Systems	52
114	Seidl Computer Engineering	125
217	Semantic Microsystems	33
85	SemiDisk Systems	99
212	Sextant	67
202	Simon & Schuster Computer Books	42
219	Simon & Schuster Computer Books	41
83	Soft Advances	95
*	Soft Focus	83
113	SoftCraft	2
88	Softaid	109
197	Software Masters	20
155	Solution Systems	56
153	Solution Systems	74
152	Solution Systems	74
147	Solution Systems	71
164	Spruce Technologies	121
172	Sunny Hill Software	112
173	TLM Systems	49
174	TLM Systems	45
175	TLM Systems	47
150	Trio Systems	37
207	Turbo Power Software	85
214	US Software	113
120	/usr/group - UniForum	69
77	UniPress Software	15
154	Vance Info Systems	36
157	Vermont Creative Software	83
112	Wendin Inc.	9
116	Wizard Systems	43
208	Wordtech Systems	21

* This advertiser prefers to be contacted directly: see ad for phone number.

Advertising Sales Offices

East Coast
Walter Andrzejewski (617) 567-8361

Midwest
Michele Beaty (317) 875-8093

Northern California/Northwest
Lisa Boudreau (415) 424-0600

Southern California/AZ/NM
Beth Dudas (714) 643-9439

Advertising Director
Shawn Horst (415) 424-0600

Apple Mac 512™

IBM PCAT™

Commodore Amiga™

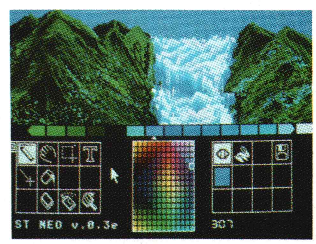
THERE'S ONLY ONE WORD FOR THESE PRICES: RIP-OFF.

Introducing the Atari 520ST personal computer system. $799.95 complete.*

Go ahead. Compare those other machines with the new Atari 520ST.™ They cost hundreds of dollars more, but you don't get much in return. That's what we call a rip-off.

For $799.95,* the 520ST comes complete with high-resolution monochrome monitor, 2-button mouse, 3.5″ disk drive, TOS™ Operating System, including GEM™ Desktop, plus Logo™ and Atari BASIC programming languages. $200 more gives you an RGB color monitor with 512 glowing colors.

Choose innovative business, entertainment, education, systems management, and integrated package software. Expand your 520ST with industry standard parallel printers, modems, MIDI controlled synthesizers and keyboards, 1 megabyte floppies, 10 MB and larger hard disks, and more. All available now. At remarkably low prices.



So, go ahead. Compare the ST system to those other guys. Only Atari gives you so much. For so little.

For the dealer nearest you, write Atari Corp., Customer Services, 1196 Borregas Ave., Sunnyvale, CA 94086.

	ATARI™ 520ST	IBM™ PCAT™	APPLE™ Macintosh™	COMMODORE™ AMIGA™
Price	$799	$4675	$2795	$1795
CPU Speed MHz	68000 8.0	80286 6.0	68000 7.83	68000 7.16
Standard RAM	512K	256K	512K	256K
Number of Keys	95	95	59	89
Mouse	Yes	No	Yes	Yes
Screen Resolution (Non-Interlaced Mode) Color Monochrome	 640 x 200 640 x 400	 640 x 200 720 x 350**	 None 512 x 342	 640 x 200*** 640 x 200***
Color Output	Yes	Optional	None	Yes
Number of Colors	512	16	None	4096
Disk Drive	3.5″	5.25″	3.5″	3.5″
Built-in Hard Disk (DMA) Port	Yes	Yes	No	No
MIDI Interface	Yes	No	No	No
No. of Sound Voices	3	1	4	4

**With optional monochrome board (non bit-mapped)
***Interlace Mode – 640 x 400

*Plus applicable local taxes. $999.95 with color monitor. All prices are manufacturer's suggested retail list.

ATARI®

Power without the price.

IBM & PCAT are registered trademarks of International Business Machines Corp. Commodore & Amiga are trademarks of Commodore Electronics LTD. Apple & Macintosh are trademarks of Apple Computer, Inc. GEM is a trademark of Digital Research, Inc. Atari, TOS & Logo are trademarks of Atari Corp.

Circle **no. 224** on reader service card.

SAVE OVER 30% ON OUR GIFT PACKS!
60-DAY MONEY-BACK GUARANTEE

How Borland's Three New Holiday Packs Will Fill Your Stocking Without Emptying Your Piggybank.

Three special packs with dazzling discounts that will help get you into a Holiday mood. You can get some of Turbo, most of Turbo, or all of Turbo—including the two newest members of the Turbo family, Turbo GameWorks™ and Turbo Editor Toolbox.™ You also get our unmatched 60-day money-back guarantee, quality products that aren't copy-protected.

TURBO NEW PACK $95.00.

You get the two exciting new members of the Turbo Pascal family,

- TURBO GAMEWORKS, Chess, Bridge, and Go-Moku, complete with source code and a 200-page manual.
- TURBO EDITOR TOOLBOX, all the building blocks to make your own editors and word processors, complete with source code and a 200-page manual.

TURBO HOLIDAY PACK $125.00.

You get all three of the Turbo family classics for only $125.00 (about a 30% discount). Turbo Pascal 3.0 and Turbo Tutor and Turbo Data-Base Toolbox—all for just $125.00.

- TURBO PASCAL combines the fastest Pascal compiler with an integrated development environment.
- TURBO TUTOR teaches you step-by-step how to use Turbo Pascal with commented source code for all program examples on diskette.
- TURBO DATABASE TOOLBOX offers three problem-solving modules for your Turbo Pascal programs: Turbo Access, Turbo Sort, and GINST, which generates a ready-to-run installation program that lets you forget about adapting your software to specific terminals.

TURBO HOLIDAY JUMBO PACK $245.00.

This is it—the whole thing, the entire Turbo family including its two newest members. You get:

- Turbo Pascal
- Turbo Graphix Toolbox
- Turbo Tutor
- Turbo DataBase Toolbox
- Turbo GameWorks
- Turbo Editor Toolbox

NEW!

and you pay only $245.00 for all six! Which means that you're getting everything at only about $40 a piece. Quite a holiday deal. (And if you already own one or several members of the Turbo family, be creative—nothing can stop you from buying the Jumbo Pack, picking out the ones you already have and giving the rest as holiday gifts to family and friends. At these prices you can afford to give to others and to yourself.) Speaking of Holidays, this offer lasts until March 31, 1986. (At Borland, we like to make the Holidays last.)

NEW! TURBO GAMEWORKS $69.95.

Our new Turbo GameWorks offers games you can play and replay without Turbo Pascal or revise and rewrite with Turbo Pascal 3.0. We give you the source code, the manual, the diskettes and the competitive edge. Chess, Bridge and Go-Moku. State-of-the-art games that let you be player, referee, and rules committee all at once because you have the Turbo Pascal source code. Learn exactly how the games are made—so you can go off and make your own. And Turbo GameWorks is the only quality game you can buy that is not copy-protected. Sold separately, only $69.95. (Just $47.50 if you buy the Turbo New Pack.)

NEW! TURBO EDITOR TOOLBOX $69.95.

Build your own word processor—for only $69.95! You get ready-to-compile source code, a full-featured word processor that looks and acts like WordStar™ and a 200-page manual that tells you how to integrate the editor procedures and functions into your programs. With Turbo Editor Toolbox you can have the best of all word processors. You can make WordStar behave like Multi-Mate. Support windows just like Microsoft's Word. And do it as fast as WordPerfect does it. Incorporate your new "hybrids" into your programs to achieve incredible control and power. Sold separately, only $69.95. (If you buy the Turbo New Pack, the price drops to just $47.50.)

NOT COPY-PROTECTED

Holiday Gift Packs, Turbo GameWorks™ & Turbo Editor Toolbox™

Available at better dealers nationwide. Call (800) 556-2283 for the dealer nearest you. To order by Credit Card call (800) 255-8008, CA (800) 742-1133.

The holiday packs include an upgrade coupon for both options so you get BCD and 8087 support for $39.95 (regularly $55.00).

Carefully describe your computer system!

Mine is: — 8-bit — 16-bit
I use: — PC-DOS — MS-DOS
— CP/M-80 — CP/M-86

My computer's name and model is: ______

The disk size I use is: ☐ 3½" ☐ 5¼" ☐ 8"

Name: ______
Shipping Address: ______
City: ______
State: ______ Zip: ______
Telephone: ______

H11

NOTE: Turbo Editor Toolbox and Turbo GameWorks are available for the IBM PC and true-compatibles using Turbo Pascal 3.0 ONLY.

*Gift Pack Offers Last Until March 31, 1986

		Quantity
*Turbo Holiday Jumbo Pack	$245.00	___
*Turbo Holiday Pack	$125.00	___
*Turbo New Pack	$95.00	___
Pascal	$69.95	___
Pascal w/8087	$109.90	___
Pascal w/BCD	$109.90	___
Pascal w/8087 and BCD	$124.95	___
Turbo DataBase	$54.95	___
Turbo Graphix	$54.95	___
Turbo Tutor	$34.95	___
Turbo Editor	$69.95	___
Turbo GameWorks	$69.95	___

These prices include shipping to all U.S. cities. All foreign orders add $10 per product ordered.

Amount: (CA add 6% tax) ______

Payment: VISA MC Check Bank Draft

Credit Card Expiration Date: ___ / ___

Card # ______

COD's and Purchase Orders will not be accepted by Borland International. California residents: add 6% sales tax. Outside USA: add $10 and make payment by bank draft, payable in U.S. dollars drawn on a U.S. bank.

BORLAND INTERNATIONAL
4585 Scotts Valley Drive, Scotts Valley CA 95066
Phone (408) 438-8400 Telex 172373

Circle no. 161 on reader service card.